FH JOANNEUM
University of Applied Sciences

Optimierung von Datenströmen zwischen Frontend und Backend: gRPC im Vergleich zu REST und GraphQL

Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science in Engineering (BSc)
eingereicht am
Fachhochschul-Studiengang **Mobile Software Development**

FH JOANNEUM (University of Applied Sciences), Kapfenberg

Betreuer/in: DI MA Michael Ulm
Eingereicht von: Michael Mühlberger

September 2025

Abstract

While gRPC is already considered a well-known and used API approach for service-to-service communication in microservice architecture with high performance requirements, gRPC-Web is still less common on the web. The performance benefits of gRPC are primarily due to binary Protocol Buffers (Protobuf) serialization format and HTTP/2. Since most browsers do not support the full set of HTTP/2 features, gRPC-Web cannot fully leverage these advantages. Therefore, REST and GraphQL still remain the dominant web API approaches for web clients. The goal of this Bachelor thesis is to evaluate how gRPC-Web performs compared to REST and GraphQL in consideration of latency, efficiency and resource usage and to identify scenarios in which using gRPC-Web for frontend-backend communication is suitable. The methodology combines a theoretical analysis with an experimental evaluation using a custom-built prototype. The prototype implements several services that cover typical web payloads (for example text or media data) and supports tests both in a browser-based web client and a microservice scenario. The measurement series includes single and parallel requests, cross-browser comparison and a contrast between browser-based frontend-backend communication and service-to-service communication in the microservice scenario. Measurements show that, in the browser context, gRPC-Web is not a viable performance-oriented replacement for REST or GraphQL. Especially for large media payloads, REST outperforms both gRPC-Web and GraphQL. Although the theoretical analysis showed Protocol Buffers as more efficient than JSON, these advantages were not measurable. This is due to the limited availability of HTTP/2 features in browsers and the additional proxy/translation layer which is required by gRPC-Web. Another disadvantage compared to REST and GraphQL is that gRPC-Web is harder to learn and lacks in regard of existing documentation, tools and community support. Overall, gRPC still remains the best option for service-to-service communication in microservice architectures, particularly under high request rates and high performance requirements. The use of gRPC-Web in browsers only makes sense, when an existing gRPC backend needs to be exposed to browsers. Improved browser support for HTTP/2 and HTTP/3 could change this assessment in the future.

Zusammenfassung

Während gRPC bereits als etablierter Standard für die Service-zu-Service Kommunikation in Microservice-Architekturen gilt, ist gRPC-Web im Web-Kontext weniger verbreitet. Die Leistungsvorteile von gRPC basieren insbesondere auf der binären Protobuf-Serialisierung und HTTP/2. Da Browser wichtige HTTP/2-Features nur eingeschränkt unterstützen, kann gRPC-Web diese Vorteile nicht vollständig nutzen. Im Browserumfeld dominieren daher weiterhin REST und GraphQL als etablierte Web-API-Ansätze. Diese Bachelorarbeit befasst sich mit den Fragen, wie sich gRPC-Web im Vergleich zu den etablierten Web-API-Technologien REST und GraphQL bezogen auf Latenz, Effizienz und Ressourcennutzung in der Frontend-Backend-Kommunikation zwischen einem Web-Client und einem Web-Server auswirkt und unter welchen Bedingungen der Einsatz von gRPC-Web für die Frontend-Backend-Kommunikation sinnvoll ist. Für die Beantwortung dieser Fragen wird ein kombinierter methodischer Ansatz gewählt: eine theoretische Analyse und eine experimentelle Evaluation anhand eines eigens entwickelten Prototyps, der die End-zu-End Latenz aus Sicht des Frontend Clients misst. Der Prototyp implementiert mehrere Services, die typische Web-Payloads (wie Text- und Mediendaten) abdecken, und Tests sowohl im Browser (Web-Client) als auch in einem Microservice-Szenario (Konsolen-Client). Basierend darauf werden Messreihen mit Einzel- und Parallelabfragen durchgeführt, verschiedene Browserumgebungen verglichen und die Frontend–Backend-Kommunikation im Browser der Service-zu-Service-Kommunikation im Microservice-Szenario gegenübergestellt. Die Messungen deuten darauf hin, dass gRPC-Web im Browserumfeld nicht mit den etablierten Web-API-Ansätzen REST und GraphQL konkurrenzfähig ist. Die in der Theorie erwarteten Performanceverbesserungen durch Protobuf konnten durch die Prototypen im Browser nicht bestätigt werden. Grund dafür sind die fehlenden HTTP/2-Features im Browser und der zusätzliche Übersetzungsschritt von gRPC zu gRPC-Web. Zusätzlich ist die Erlernbarkeit von gRPC-Web im Vergleich zu etablierten Technologien wie REST komplexer und es stehen weniger Dokumentation, Tools und Community-Ressourcen zur Verfügung. Vor allem bei großen Binärdaten erwies sich REST im Browser als robuster. Insgesamt empfiehlt sich gRPC weiterhin vor allem für Service-zu-Service Szenarien mit hoher Requestfrequenz, während eine Implementierung von gRPC-Web nur dann sinnvoll ist, wenn bereits ein gRPC-basiertes Backend besteht. Bei verbesserter Browser-Unterstützung für HTTP/2 oder HTTP/3 könnte sich dies künftig ändern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Bei Echtzeitanwendungen wie Chat-Applikationen oder Streaming Diensten werden oft große Datenmengen ausgetauscht. Dabei ist eine effiziente Übertragung der Daten essenziell, damit die Latenz so klein wie möglich bleibt. Eine Methode um dies in einem Softwareprojekt zu implementieren ist das gRPC Remote Procedure Call (gRPC) Framework, indem die Daten binär kodiert mittels dem Hypertext Transfer Protocol (HTTP)/2 Protokoll übertragen werden. ((gRPC Authors, 2025a)) Während sich diese Strategie für Datenströme in Backend-Architekturen und insbesondere in Microservice-Umgebungen bereits etabliert hat, ist eine nahtlose Implementierung von gRPC in Frontendanwendungen in Web-Browsern noch nicht möglich. Hauptgrund dafür, sind fehlende HTTP/2-Features in den Browsern (z. B. bidirektionales Streaming) welche eine Voraussetzung für eine Implementierung von gRPC darstellen. Eine Abhilfe dafür ist gRPC-Web, eine angepasste Variante von gRPC, die dafür entwickelt wurde, gRPC auch in Webbrowsern nutzbar zu machen. Während durch die Verwendung von gRPC-Web einige Features verloren gehen, bleiben auch wichtige Vorteile, wie etwa die Nutzung von Protocol Buffers, erhalten. ((Brandhorst, 2019)) In der Praxis wird eine gRPC-Übertragung in der Regel nur verwendet, wenn eine effiziente und schnelle Kommunikation besonders wichtig ist. Ziel der Bachelorarbeit ist es zu untersuchen, inwiefern sich gRPC beziehungsweise gRPC-Web auch in Frontend-Anwendungen sinnvoll einsetzen lassen. Hierfür werden verschieden Application Programming Interface (API)-Übertragungsarchitekturen (Representational State Transfer (REST) und Graph Query Language (GraphQL)), welche sich bereits in der Frontend-zu-Backendkommunikation etabliert haben, mit gRPC und gRPC-Web überprüft und

verglichen. ((Red Hat, Inc., 2020)) Der Schwerpunkt liegt auf den Kriterien Latenz, Effizienz und den generellen Vor- und Nachteilen, die beim Einsatz der jeweiligen Technologien zum Vorschein kommen. Die Bewertung erfolgt sowohl auf theoretischer Grundlage als auch anhand einer praktischen Untersuchung.

1.2 Forschungsfragen

Aus der dargestellten Situation und den Einbußen welche mit der Implementierung von gRPC in Webanwendungen einhergehen, ergibt sich die Motivation, gRPC / gRPC-Web mit den etablierten API-Ansätzen für die Frontend- zu Backendkommunikation zu vergleichen. Dafür werden im Rahmen der Bachelorarbeit folgende Forschungsfragen untersucht:

- Wie wirkt sich die Verwendung von gRPC bzw. gRPC-Web im Vergleich zu REST und GraphQL auf die Latenz, Effizienz und Ressourcennutzung in der Frontend-Backend-Kommunikation aus?
- Unter welchen Bedingungen ist der Einsatz von gRPC für die Frontend-Backend-Kommunikation sinnvoller als REST oder GraphQL?

Diese Forschungsfragen bilden die Grundlage für die theoretische Analyse sowie die praktische Untersuchung der genannten API-Technologien. Ziel ist es, anhand dieser Fragen sowohl die Stärken als auch die Schwächen von gRPC-Web in Webumgebungen zu identifizieren.

1.3 Hypothese

Für eine strukturiertere Herangehensweise werden folgende Hypothesen formuliert, die die erwarteten Ergebnisse darstellen:

- Es wird angenommen, dass die Nutzung von gRPC im Vergleich zu REST und GraphQL die Latenz reduziert und die Effizienz erhöht. Vor allem durch das Verwenden von Protocol Buffers, ein binäres Serialisierungsformat welches auch bei gRPC-Web verwendet wird, ist eine Effizienzsteigerung im Gegensatz zum klassischen JSON-Format zu erwarten, wodurch sich auch beim Datenaustausch bei praktischen Messungen eine geringere Latenz zeigen sollte.
- Für gRPC-Web wird angenommen, dass sich zwar performancetechnische Vorteile ergeben werden, sich die Implementierung in Projekten der Frontendanwendung jedoch als komplexer gestalten wird. Demnach wird der Einsatz von

gRPC-Web vor allem bei Anforderungen mit hohen Datenmengen sinnvoll sein.

1.4 Methodik

Folgende methodische Herangehensweise wird für die Beantwortung der Forschungsfragen angewendet:

Theoretische Analyse:

- Systematische Literaturrecherche der theoretischen Grundlagen.
- Ermittlung des aktuellen Stands der Technik auf Basis wissenschaftlicher Publikationen und relevanter Industriestandards.
- Vergleich sowie Gegenüberstellung der Vor- und Nachteile der betrachteten API-Architekturen.

Erstellen eines Prototyps:

- Experimentelle Methodik: Im praktischen Abschnitt der Arbeit wird ein Prototyp entwickelt, der die ausgewählten API-Technologien implementiert.
- Auf Basis dieses Prototyps werden Messreihen durchgeführt, welche zur Beantwortung der Forschungsfragen beitragen.

Basierend auf den theoretisch und praktisch ermittelten Daten, werden anschließend die Forschungsfragen beantwortet.

Kapitel 2

Theoretische Grundlagen

2.1 Frontend und Backend

Die Arbeit untersucht die Kommunikation zwischen Frontend zu Backend Komponenten, im technischen Kontext sind diese Begriffe wie folgt definiert:

> *"Das Frontend ist das, was die Benutzer sehen, und enthält visuelle Elemente wie Schaltflächen, Kontrollkästchen, Grafiken und Textnachrichten. Es ermöglicht den Benutzern, mit der Anwendung zu interagieren. Das Backend sind die Daten und die Infrastruktur, die dafür sorgen, dass die Anwendung funktioniert. Es speichert und verarbeitet Anwendungsdaten für die Benutzer."*
>
> [Amazon Web Services, Inc., 2024]

Wie aus dieser Definition ersichtlich, beinhaltet das Frontend den benutzerspezifischen Teil und das Backend den Datenverarbeitungsteil. In Anlehnung an die Server-Client-Architektur wird im Zuge der Bachelorarbeit die Frontend Komponente auch als „Client“ und die Backend Komponente als „Server“ bezeichnet.

2.2 Serialisierungsformate

Bei der Übertragung zwischen dem Frontend und Backend werden Daten ausgetauscht. Da es eine Vielzahl an Formaten gibt, mit denen die jeweiligen Datenobjekte für die Übertragung serialisiert werden können, werden die in der Arbeit verwendeten Serialisierungsformate anschließend erläutert.

2.2.1 JSON

JavaScript Object Notation (JSON) ist ein weit verbreitetes, textbasiertes Datenformat, das vor allem wegen seiner einfachen Lesbarkeit und der breiten Unterstützung in vielen Programmiersprachen Verwendung findet. Das Format basiert auf einer Schlüssel-Wert-Paar-Struktur mit einfacher Syntax (Klammern, Doppelpunkte, Kommas) und kann folgende Datentypen annehmen:

- String (Zeichenkette)
- Number (Zahl)
- Boolean (true/false)
- Array (Liste)
- Object (Objekt mit weiteren Schlüssel-Wert-Paaren)
- null (leerer Wert)

(Ecma International, 2017)

Beispiel eines JSON-Objekts:

```
{
 "id": 1,
 "name": "Alice",
 "email": "alice@example.com",
 "isActive": true
}
```

JSON ist der Standard für REST und GraphQL Schnittstellen (GraphQL Foundation, 2025; Microsoft Azure Architecture Center, 2025).

2.2.2 Protocol Buffers

Protocol Buffers sind ein von Google entwickeltes Serialisierungsformat. Es handelt sich hierbei um ein binäres Serialisierungsformat, das entwickelt wurde um möglichst effizient, mit hoher Performance und mit so wenig Overhead wie möglich (ohne Whitespaces oder Satzzeichen wie bei JSON) Daten zu übertragen und zu verarbeiten. Protocol Buffers sind plattformunabhängig und mit den meisten gängigen Programmiersprachen kompatibel.

Ein zentraler Bestandteil von Protocol Buffers sind die plattformunabhängigen proto Files, die für die Erzeugung definiert werden müssen.

Mit den definierten proto Files können anschließend mit einem Protobuf-Compiler-Tool (z.B. protoc) Datenobjekte der jeweils eingesetzten Programmiersprache generiert werden.

Aufbau der proto Files

Beispiel einer Protocol Buffers Definition mit den Datenobjekten Person und PersonRequest und dem Service PersonService:

```
syntax = "proto3";

service PersonService {
 rpc GetPerson (PersonRequest) returns (Person);
}

message Person {
 string name = 1;
 int32 id = 2;
 string email = 3;
}

message PersonRequest {
 int32 id = 1;
}
```

Hauptbestandteil der proto Files sind Messages und Services. Messages definieren die Struktur der zu übertragenden Nachricht. Jedes Feld besteht aus einem Namen, dem jeweiligen Datentyp und einer Nummer. Die Nummer beschreibt an welcher Stelle sich das jeweilige Attribut befindet. Die Nummerierung der Felder ist wichtig für die Serialisierung. In Protocol Buffers können unter anderem die gängigsten Datentypen wie int32, int64, float, double, bool sowie string und bytes verwendet werden. Services geben an, welche Dienste vom Server bereit gestellt werden und welche Datentypen als Parameter bei Aufruf übermittelt und als Rückgabe zurückgegeben werden. Services müssen auf der Serverseite implementiert werden (Google, 2024).

2.2.3 Blob

Für das Übertragen von großen binären Objekten (z.B. Bilder, Video, Audiodateien, Dokumente, ..) wird eine große Menge an binären Daten gesendet. Solche Dateien werden in der Webentwicklung, und vor allem im Frontend wo diese Mediendaten

verarbeitet werden sollen, oft als Binary Large Object (Blob) bezeichnet (World Wide Web Consortium (W3C), 2024).

2.3 Transportprotokolle

Die eigentlichen Daten, die in Form von Serialisierungsformaten zwischen Frontend und Backend ausgetauscht werden, werden mithilfe von Transportprotokollen von dem Client an den Server, und umgekehrt, übermittelt. Das Hypertext Transfer Protocol (HTTP) ist dabei für moderne Webanwendungen das zentrale und verbreitetste Transportprotokoll. Es gibt verschiedene Versionen von HTTP. Im Kontext der Bachelorarbeit wird ausschließlich HTTP/2 verwendet. HTTP funktioniert mittels eines Request-Response Prinzips und überträgt die Daten mittels TCP.

HTTP/1.1 ist nach wie vor weit verbreitet. Es wird von allen Webbrowsern ohne Einschränkungen unterstützt und bildet die Grundlage für die API-Architekturen REST und GraphQL. Diese Version weist jedoch einige Schwächen auf, so können zum einen nicht mehrere Requests/Responses gleichzeitig über eine TCP Verbindung durchgeführt werden und der Server ist nicht in der Lage von sich aus zusätzliche Ressourcen an die Clients zu senden.

Neben Performanceverbesserungen und anderen zusätzlichen Features wurden diese Schwächen mit HTTP/2 behoben. Die wichtigsten Features von HTTP/2 beinhalten:

- **Multiplexing:** Das wichtigste Merkmal von HTTP/2 ist Multiplexing, durch das es möglich ist, mehrere parallele Anfragen und Antworten über nur eine einzige TCP-Verbindung durchzuführen.
- **Server-Push Verfahren:** Ermöglicht es dem Server von sich selbst aus Ressourcen an Clients zu senden, ohne dass zuvor ein Request notwendig ist.
- **Header-Komprimierung:** Geringerer Overhead durch effizientere Übertragung sich wiederholender Headerinformationen.
- **Binäres Protokoll:** Statt dem textbasierten Protokoll wird nun ein binäres Protokoll verwendet, was zur Effizienzsteigerung führt.
- **Trailers:** Möglichkeit, zusätzliche Header-Felder am Ende einer Übertragung zu senden.

Aufgrund der genannten Verbesserungen wird HTTP/2 besonders in Systemen, in denen eine schnelle und effiziente Datenübertragung wichtig ist, verwendet (Thomson & Bishop, 2022).

Zwar unterstützen aktuelle Webbrowser HTTP/2 grundsätzlich, jedoch sind bestimmte Funktionen wie zum Beispiel echtes bidirektionales Streaming (gleichzeitiges Senden und Empfangen von Daten durch Client und Server), in Browserumgebungen nicht beziehungsweise nur bedingt möglich. Dadurch können bei der Kommunikation zwischen Frontend (Webbrowser) und Backend nicht alle Potenziale von HTTP/2 ausgeschöpft werden (Microsoft Docs, 2025).

2.4 API-Technologien

Eine API ist eine Schnittstelle eines Softwaremoduls, welche es ermöglicht, dass das jeweilige Softwaresystem mit einem anderen System kommunizieren kann. Eine Web-API ist eine API die von Web-Servern zur Verfügung gestellt werden. Bei Web-Servern findet die Kommunikation standardmäßig mit einem HTTP bzw. Hypertext Transfer Protocol Secure (HTTPS) Protokoll statt. Technologien wie REST, GraphQL oder gRPC definieren jeweils Konventionen und Standards, wie etwa den Aufbau von Response/Request, Art der Serialisierungsobjekte oder Art des Transportprotokolls, die neben der technischen Kompatibilität für die Kommunikation auch den Entwicklern hilft, sich effizient in API-Systemen zurecht zu finden (Red Hat, Inc., 2020).

2.4.1 REST

Der REST-Architekturstil wurde im Jahr 2000 von Roy Fielding im Rahmen seiner Dissertation definiert, mit dem Ziel eine bis dato einfache Alternative für die Kommunikation zwischen zwei Systemen zu schaffen.

Infrastruktur

Eine API-Schnittstelle ist nach Definition nur dann im REST Stil umgesetzt, falls folgende 5 Eigenschaften erfüllt werden:

- **Client-Server:** Das System muss in einer Server-Client Architektur umgesetzt werden. Bestehend aus einem Server, der einen Dienst oder Daten bereitstellt und einem Client, welcher diese Dienste nutzen kann. Server und Client kommunizieren mittels Nachrichten, indem mittels eines Request-Response Prinzips der Client eine Anfrage (Request) schickt und der Server eine Antwort (Response) zurückliefert.
- **Zustandslosigkeit:** In jeder Nachricht, die zwischen Server-Client verschickt werden, müssen alle Informationen enthalten sein, damit der jeweilige Kommu-

nikationspartner die Nachricht verarbeiten kann. Es ist nicht erlaubt Zustandsinformationen zwischen zwei Nachrichten zu speichern.

- **Caching:** HTTP Caching soll benutzt werden. Hierbei werden Daten in einem Cache gespeichert und bei nochmaligem Abfragen vom Cache abgefragt, um unnötige Serveranfragen und Datenübertragungen zu vermeiden.
- **Einheitliche Schnittstelle:** Eine einheitliche Schnittstelle ist wie folgt definiert:
  1. **Adressierbarkeit von Ressourcen:** Jede Ressource im System ist eindeutig über eine URI adressierbar.
  2. **Repräsentationen zur Veränderung von Ressourcen:** Ressourcen können durch verschiedene Formate (wie JSON) repräsentiert und über Standard-HTTP-Methoden verändert werden.
  3. **Selbstbeschreibende Nachrichten:** Jede Nachricht enthält alle notwendigen Informationen, damit der Empfänger sie interpretieren und verarbeiten kann.
  4. **„Hypermedia as the Engine of Application State“ (HATEOAS):** Bei HATEOAS stellt der Server dem Client Links zur Verfügung, mittels dem der Client auf weitere Ressourcen zugreifen kann.
- **Mehrschichtige Systeme:** Systeme sollen mehrschichtig aufgebaut sein, die Kommunikationsparner müssen aber jeweils immer nur die oberste Schicht kennen.
- **Code on Demand (optional):** Im Bedarfsfall kann der Server dem Client Code zur Verfügung stellen. Diese Eigenschaft ist für einen REST Architekturstil jedoch nur optional.

(Fielding, 2000)

Viele Web Dienste, erfüllen bereits viele dieser Eigenschaften wodurch sich REST zu einem beliebten Standard entwickelt hat.

Kommunikation zwischen Client-Server:

Die Kommunikation zwischen dem Client und Server findet bei REST fast ausschließlich über HTTP/HTTPS statt, wobei laut Definition kein spezifisches Transportprotokoll vorgeschrieben wird. Im Zuge der Arbeit wird ausschließlich HTTPS verwendet.

HTTP stellt für den Zugriff auf Ressourcen eine Reihe an Methoden zur Verfügung. Die wichtigsten HTTP-Methoden sind:

HTTP Methode	**Beschreibung**
GET	GET fordert eine Ressource vom Server an. Der Zustand des Dienstes bzw. der Ressource wird dabei nicht verändert.
POST	Fügt in den meisten Fällen eine neue Ressource ein. Kann aber auch verwendet werden, um Methoden zu realisieren, die durch keine andere HTTP-Methode abgedeckt werden. Ändert den Zustand des Servers.
PUT	Erstellt eine Ressource neu oder ersetzt eine bestehende Ressource vollständig.
PATCH	Nimmt partielle Änderungen an einer bestehenden Ressource vor.
DELETE	Löscht eine bestehende Ressource.

Der Zugriff auf die Ressourcen erfolgt bei REST jeweils über die URL (Group, 2022).

2.4.2 GraphQL

GraphQL ist eine Open Source Datenabfrage- und Manipulationssprache und wurde im Jahr 2015 von Facebook veröffentlicht. Der Standard wurde entwickelt, um eine bessere Alternative für REST-Architekturen und SQL bereitzustellen, da bei REST-Anwendungen ein vordefinierter Satz von Daten an den Client zurückgeliefert wird. Vor allem bei mobilen Anwendungen führte diese Einschränkung zu Problemen, da oft zu viele bzw. zu wenig Daten übermittelt wurden, was zu einem unnötigen Übertragungsvolumen führt. GraphQL erlaubt es dem Client gezielt die benötigten Daten abzufragen. Außerdem können Daten, welche in mehreren Datenobjekten verschachtelt sind, effizient abgebildet werden. Diese Eigenschaften machen GraphQL im Kontext der Datenabfrage besonders effizient und flexibel, da somit keine unnötige Datenübertragung stattfindet.

Unterstützte Operationen

Queries (schreibend):

Queries definieren die exakten Daten die vom Client angefordert werden. Die Daten werden in der selben Struktur an den Client zurückgesendet.

Mutations (manipulierend):

Mit Mutations können Daten manipuliert werden, ähnlich wie POST, PUT/PATCH oder DELETE Funktionen bei REST. Mutations beinhalten Variablen welche vom GraphQL Server verarbeitet werden und eine Definition der erwarteten Struktur der Antwort.

Subscriptions:

Erlauben Live Updates von dem Server an den Client. Die Definition legt fest, in welcher Struktur die Nachrichten an den Client übermittelt werden.

Technische Umsetzung

Technisch werden GraphQL Requests und Responses in einem JSON Format übertragen und die Kommunikation findet über das HTTP-Protokoll statt. Subscriptions werden jedoch häufig über WebSockets realisiert. Außerdem ist es Konvention, dass typischerweise alle Operationen (Queries, Mutations, Subscriptions) über einen einzigen Endpunkt (/graphql) laufen, üblicherweise über HTTP POST. Die Anfrage wird dabei als String im sogenannten „GraphQL Query Language“-Format an den Server geschickt. Die Antwort enthält die angeforderten Daten innerhalb eines „data“ Feldes und im Fehlerfall ein „errors“-Feld in dem die jeweiligen Fehler und Details angeführt werden (GraphQL Foundation, 2025).

2.4.3 gRPC

gRPC ist ein im Jahr 2015 veröffentlichtes Open Source Framework mit dem Remote Procedure Calls (RPC) durchgeführt werden können. Die Grundidee von gRPC geht darauf zurück, dass Google eine moderne und performante Technologie entwickeln wollte, das auf einem bereits intern eingesetztem RPC Framework names Stubby basiert. Stubby wurde ursprünglich innerhalb von Google entwickelt und eingesetzt, um die Kommunikation zwischen einer Vielzahl an Microservices in unterschiedlichen verteilten Systemen effizient umzusetzen. gRPC knüpft an dieser Technologie an und wurde, als Open Source Nachfolger von Stubby veröffentlicht, um eine performante und sprachübergreifende Lösung für die Interprozesskommunikation bereitzustellen (gRPC Authors, 2025a).

RPC: RPC beschreibt ein Kommunikationsmodell, bei dem ein Programm eine Methode auf einem anderen entfernten System aufruft, als wäre sie lokal vorhanden. Im Unterschied zu REST oder GraphQL sind RPC Calls Methoden (Methodenname, Parameter und Rückgabewert) und nicht Ressourcen orientiert, wodurch die Schnittstelle stärker an der Logik der Anwendung angelehnt ist (AWS, 2025).

Einsatzgebiete von gRPC: Hauptanwendung findet gRPC in der Kommunikation von Microservice-Architekturen und in der inter-backend-Kommunikation. Microservices sind kleine, eigenständige Dienste, die jeweils eine klar abgegrenzte Aufgabe erfüllen und unabhängig voneinander entwickelt werden, jedoch häufig in hoher Frequenz und mit großen Datenmengen miteinander kommunizieren. Hier ist es wichtig, dass die Daten performant zwischen den verschiedenen Services hin und her geschickt werden. gRPC kann jedoch auch für die Kommunikation zwischen Browsern bzw. Mobilgeräten und Backend-Services genutzt werden (gRPC Authors, 2025a).

Technische Grundlagen: Die Kommunikation von gRPC wurde speziell auf Basis von HTTP/2 entwickelt und findet demnach standardmäßig über das HTTP/2 Protokoll statt. (Mugur Marculescu, 2015). Außerdem werden für die Serialisierung Protocol Buffers verwendet, welche durch die binäre Struktur zu einer weiteren Verbesserung der Performance und Effizienz führen (Google, 2024).

2.4.4 gRPC-Web

Obwohl gRPC ursprünglich für die Interprozesskommunikation in Microservices entwickelt wurde, ist es auch möglich gRPC mit Einschränkungen für die Kommunikation zwischen Web-Browsern und Backend Services zu verwenden. Da in Web Browsern nicht alle Funktionen von HTTP/2 zur Verfügung stehen, kann für die Kommunikation in diesem Szenario nicht gRPC direkt verwendet werden. Für diesen Anwendungsfall wurde ein eigenes Protokoll namens gRPC-Web konzipiert.

Folgende Funktionen von gRPC stehen dadurch nicht zu Verfügung:

- **Bidirektionales Streaming:** Im Browser ist kein echtes bidirektionales Streaming möglich. Es werden nur Unary RPCs und Server-Streaming unterstützt.
- **Metadaten und Header-Kompression:** Nicht alle gRPC-Metadaten können übertragen werden, und eine HTTP/2-Header-Kompression wird nicht unterstützt.
- **Trailers:** gRPC-Web kann keine echten HTTP/2-Trailer verwenden. Status- und Fehlercodes werden daher in der Response umkodiert.

Folgende Eigeschaften bleiben bei der Verwendung des gRPC-Web Protokolls erhalten:

- **Protocol Buffers als effizientes Datenformat**
- **Codegenerierung für Server und Client**
- **Typensicherheit**
- **Server-Side Streaming**

Oft wird gRPC-Web über einen Proxy (zum Beispiel Envoy oder gRPC-Web-Proxy) an einen regulären gRPC-Server weitergeleitet. Dies muss gemacht werden, da gRPC-Web zwar HTTP/1.1 oder eingeschränktes HTTP/2 (wie es im Browser verfügbar ist) verwendet, der eigentliche gRPC Server jedoch auf vollem HTTP/2 basiert. Ein solcher Proxy übernimmt die „Übersetzung“ zwischen gRPC-Web und gRPC (Backend) (Brandhorst, 2019). Für diese Übersetzung gibt es verschiedene gängige Varianten:

1. **Envoy Proxy:** Envoy ist ein moderner Proxy, der nativ gRPC-Web unterstützt. Er führt die „Übersetzung“ durch und leitet sie intern an den gRPC-Server weiter.
    - Vorteil: Skalierbar und performant
    - Nachteil: Zusätzlicher Deployment Aufwand (eigene Proxy-Instanz erforderlich)

2. **gRPC-Web-Proxy:** Ein Node.js-basierter Proxy, der ebenfalls als Brücke zwischen Browser und gRPC-Backend dient.
    - Vorteil: Schnell einzurichten
    - Nachteil: Nicht so leistungsfähig als Envoy
3. **Direkte Serverintegration in ASP.NET Core:**
    - Vorteil: Einfaches Setup und kein Proxy nötig. Ideal für .NET-Umgebungen
    - Nachteil: Nicht so flexibel für komplexe Architekturen

Im Rahmen des implementierten Prototyps für diese Arbeit wurde die direkte Serverintegration in ASP.NET Core gewählt (gRPC Authors, 2025b; Microsoft Docs, 2025).

Da es in der Arbeit um die Kommunikation zwischen Web Browsern und Backend-Services geht, wird in den folgenden Kapiteln vor allem ein Fokus auf das gRPC-Web Protokoll gelegt.

2.5 Begriffe

2.5.1 End-zu-End Latenz

Unter dem Begriff Latenz versteht man in der Netzwerktechnik die Verzögerung, die zwischen dem Absenden einer Anfrage (Request) und dem Eintreffen der dazugehörigen Antwort (Response) auftritt (Amazon Web Services, Inc., 2025). Im Rahmen der Bachelorarbeit bezieht sich der Begriff jedoch auf die End-zu-End Latenz. Also jener Zeit ab dem Senden des Requests von dem Client bis zur vollständigen Verarbeitung und Bereitstellung der Antwort im Client.

2.5.2 Durchschnitt und Median

Zur Auswertung von Messergebnissen wurden zwei Lageparameter verwendet: der Durchschnitt (arithmetische Mittel) und der Median.

Der Durchschnitt ergibt sich aus der Summe aller gemessenen Werten, dividiert durch die Anzahl der Messungen:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Der Median bezeichnet jenen Wert, der in einer sortierten Messreihe genau in der Mitte liegt. Er teilt die Messwerte in zwei gleich große Hälften und ist dadurch robuster gegenüber Ausreißern (Ludwig-Mayerhofer, 2025).

2.5.3 Browser Engines

Jeder moderne Webbrowser besitzt eine sogenannte Browser Engine, diese Engine ist unter anderem für das Rendern von Webseiten, die Verwaltung von Netzwerkverbindungen sowie die interne Verarbeitung von Inhalten zuständig. Bekannte Browser Engines sind zum Beispiel Blink (Chrome, Edge) und Gecko (Firefox) (Chrome Developers, 2025).

Kapitel 3

Stand der Technik

Moderne Webanwendungen verwenden je nach Anforderungen eine Vielzahl an verschiedenen Technologien und Frameworks. Um zu ermitteln inwiefern sich die Implementierungen von REST, GraphQL und gRPC-Web nach aktuellem Stand unterscheiden und in welchem Szenario welche API-Technologie bevorzugt eingesetzt wird, werden aktuelle Statistiken, Artikel und wissenschaftliche Arbeiten zu dem Thema analysiert.

3.1 Industrielle Standards

3.1.1 Verbreitung der API-Technologien

Sowohl REST, GraphQL, als auch gRPC sind momentan weit verbreitete API-Architekturen, wobei der Verbreitungsgrad zwischen den Architekturen stark variiert. Bezogen auf den „State of the APIs Report“ Bericht der populärsten API-Technologien von Postman aus den Jahren 2022 und 2023, deren Ergebnisse auf einer weltweiten Umfrage unter 40.261 Entwicklern und API-Fachleuten, sowie aggregierten Daten der Postman-Plattform basieren, ergeben sich folgende Ergebnisse:

- REST ist mit Abstand die populärste API. Während die Verwendung in den letzten Jahren aufgrund von neuen Architekturen langsam abnahm, gaben 86% der Befragten an, REST zu benutzen. In den Jahren zuvor waren dies 89% (2022) und 92% (2021).
- GraphQL ist nach Webhooks momentan die drittbeliebteste API-Architektur, 29% der Befragten benutzten GraphQL. Der Trend zeigt, dass die Technologie in den letzten Jahren an Beliebtheit dazugewann. In den Jahren zuvor nutzten

28% (2022) bzw. 24% (2021) GraphQL.

- gRPC ist in dem Report auf Platz 6, 11% der Befragten benutzten die API-Technologie. Damit stagnierte gRPC im Vergleich zum Vorjahr indem ebenfalls 11% (2022) der Befragten gRPC nutzten. Im Jahr 2021 waren es noch 8%. Hier ist anzumerken, dass es nicht ersichtlich ist, wie viele der Befragten gRPC-Web benutzten. Da der Hauptanwendungsbereich von gRPC jedoch bei Anwendungen in Microservice Architekturen liegt, ist anzunehmen, dass dies nur ein sehr kleiner Bruchteil ist, und gRPC-Web derzeit nicht sehr populär ist.

  Entsprechend der Popularität liegen zu REST, GraphQL und gRPC eine dementsprechende Anzahl an Test, Debugging und Monitoring Tools vor. Für gRPC-Web wurden in Eigenrecherche kaum Tools dieser Art vorgefunden (Postman, 2022, 2023).

3.1.2 Verbreitung von Web-Frontend Frameworks

Neben API-Technologien beschäftigt sich die Arbeit vor allem mit der Kommunikation von Datenströmen aus der Sicht einer Web Frontend Anwendung. Laut der „Stack Overflow Developer Survey 2023“ gehören React, Angular und Vue.js zu den am häufigsten verwendeten Webframeworks, wobei React derzeit am weitesten verbreitet ist. Wie das Kapitel „Implementierung“ im praktischen Teil dieser Bachelorarbeit verdeutlicht, bestehen auch zu allen in der Arbeit analysierten API-Architekturen entsprechende Bibliotheken und Tools für eine Implementierung in React (Stack Overflow, 2023).

3.1.3 Einsatzbereiche der API-Architekturen in der Industrie

Um die Relevanz der untersuchten Technologien in der gegenwärtigen Industrie aufzuzeigen und für welche Einsatzbereiche sie dort verwendet werden, werden im Folgenden ausgewählte Praxisbeispiele großer internationaler Unternehmen betrachtet.

- REST gilt nach wie vor als Standard für öffentliche Web-APIs und Web-Services, und wird somit flächendeckend in der Industrie verwendet. Beispiele der Einsätze sind:
  - Führende Cloud-Anbieter wie Amazon Web Services (AWS), Microsoft Azure und die Google Cloud Platform(GCP) nutzen REST-APIs für den primären Datenzugriff für die Dienste wie Cloud-Speicherung, Datenverarbeitung, KI-Dienste, ..
  - Unternehmen oder E-Commerce-Plattformen wie eBay bieten oft umfang-

freiche REST-APIs an, damit Entwickler Daten und Funktionen der Plattform in ihren eigenen Anwendungen integrieren können. In dem Fall von eBay wären dies etwa ein Zugriff auf Produkt-, Angebots- oder Bestelldaten.

(Postman Team, 2023)

- GraphQL wurde im Jahr 2012 von Facebook entwickelt und 2015 als Open Source Projekt bereitgestellt. Der Hauptgrund dafür war, dass Facebook während der Entwicklung, deren nativen Apps einen effizienten Austausch von Daten ermöglichen wollte um das Problem von Over- und Underfetching zu vermeiden (Amazon Web Services, 2025; GraphQL Foundation, 2025).
  - PayPal nutzt GraphQL seit 2018. Grund der Umstellung war um die zuvor fragmentierte API-Landschaft zu vereinheitlichen und die Developer Experience zu verbessern. Durch den sprachunabhängigen Endpunkt wird die Entwicklung beschleunigt und die Integration für externe Händler vereinfacht.
  - Netflix nutzt seit 2021 GraphQL Microservices um Datenbanken schnell als APIs bereitzustellen zu können, um damit CRUD Anwendungen effizienter entwickeln zu können und somit die Produktivität der Teams zu erhöhen.
  - GitHub stellt seit 2016 eine öffentliche GraphQL-API zur Verfügung, damit Entwickler gezielt nur die benötigten Daten in einem einzigen Aufruf abfragen können und um die Integration mit externen Services zu vereinfachen.
  - Shopify führte 2018 GraphQL für eine Admin-API ein um Probleme mit dem Client-Server-Datenmapping bei REST zu lösen. Bei der REST API waren externe Developer bei jedem API Update gezwungen deren Code anzupassen (Charboneau, 2022).
- gRPC wurde von Google selbst entwickelt und wird dort sowohl für die interne Service-zu-Service-Kommunikation als auch in zahlreichen Google-Cloud-APIs eingesetzt. Hauptgrund für die Entwicklung war die Notwendigkeit eines leistungsfähigen, modernen und effizienten Frameworks für RPC Calls (gRPC Authors, 2025a).
  - Netflix benutzt gRPC intensiv für die interne Kommunikation zwischen seinen hunderten Microservices. Dabei wird es vor allem benutzt um Backend-

zu-Backend Aufrufe so effizient wie möglich durchzuführen (Netflix Tech Blog, 2023).

- Roblox, ein populäres Videospiel mit Millionen gleichzeitig aktiven Nutzern, nutzt gRPC und Protocol Buffers intern für die Kommunikation zwischen den Backendservices um eine hohe Performance und die Skalierbarkeit der Infrastruktur zu gewährleisten (Roblox Newsroom, 2025).
- Spotify nutzt seit 2019 gRPC in der Kommunikation zwischen seinen Backend-Services und hat damit seine früheren Eigenentwicklungen ersetzt (Kubernetes Authors, 2025).

Die Beispiele zeigen klar die Haupteinsatzgebiete der jeweiligen API-Architekturen auf. So kann abschließend gesagt werden, dass REST wegen der weiten Verbreitung und Stabilität oft für öffentliche APIs genutzt wird und sich besonders für breite Entwickler-Communities und externe Integrationen eignet. GraphQL wird benutzt, wenn flexible Datenabfragen und die Minimierung von Over-/Underfetching entscheidend sind, dies zeigt auch das Nutzen von GraphQL in komplexen Frontend Apps wie Facebook. gRPC wird hauptsächlich für interne Kommunikation mit großer Datenabfragedichte genutzt, was zeigt, dass es sich besonders für performante, hochskalierende Microservice-Architekturen und Echtzeitanwendungen eignet, in denen eine geringe Latenz wichtig ist.

3.2 Verwandte Arbeiten

Neben dem industriellen Stand der Technik ist es im Zuge der Arbeit wichtig zu ermitteln, welche wissenschaftlichen Arbeiten es bereits bezogen auf die betrachteten Forschungsfragen gibt. Es wurden mehrere vergleichende Studien, Bachelor- und Masterarbeiten sowie wissenschaftliche Artikel zu dem Thema „Vergleich von REST, GraphQL und gRPC in Systemen mit Microservice Architekturen gefunden“. Im Folgenden werden einige relevante Arbeiten davon aufgezeigt. Während zahlreiche Arbeiten die Technologien im Kontext von Microservices untersuchen, fehlen vergleichbare Arbeiten aus der Perspektive von Webanwendungen, insbesondere in Bezug auf die Kommunikation zwischen Frontend und Backend mit gRPC-Web.

3.2.1 Comparative review of selected Internet communication protocols

Die einzige Arbeit, die gefunden wurde, welche auch einen Bezug zwischen Web-Server und Web-Clients herstellt ist von Kaminski und weiteren Autoren. Die Autoren der Arbeit implementieren hierbei mehrere Webserver mit REST, gRPC, GraphQL und WebSockets in Python und dazu passende Python-Clients um die Protokolle in verschiedenen CRUD-Szenarien zu benchmarken. Wichtig hierbei ist anzumerken, dass die Arbeit zwar von „Web-Clients“ spricht, es sich im Versuch jedoch nicht um browser-basierte Frontends handelt und somit nicht gRPC-Web verwendet wurde. Die Ergebnisse der Arbeit zeigen, dass gRPC in den meisten Tests die beste Performancewerte hatte, vor allem bei mehreren gleichzeitigen Requests. REST lag im Mittelfeld und GraphQL war im Vergleich am langsamsten (Kamiński u. a., 2022).

3.2.2 Efficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems

Bolanowski et al. (2022) untersuchten die Effizienz von REST und gRPC in Microservice Architekturen mit mehreren Testszenarien. Dabei werden Daten von kleineren Payloads (kurzer String oder Integer mit wenigen Bytes), über strukturierte Daten mit ein paar hundert Bytes bis hin zu Dateien mit einigen kB oder MB gesendet. Der Server wurde mit C# .NET 5 implementiert und Messungen fanden mit JMeter und IxLoad statt. Ergebnisse zeigen, dass REST bei sehr kleinen Datenmengen im Bytes Bereich schneller ist, während gRPC bei größeren Payloads (ab einigen zehn kB) deutliche Vorteile in der Performance aufweist. Die Arbeit zeigt, dass die Wahl der Technologie von der Datenmenge abhängt. Vor allem bei datenintensiven Szenarien ist gRPC besonders gut geeignet (Bolanowski u. a., 2022).

3.2.3 Benchmarking the request throughput of conventional API calls and gRPC

Die Bachelorabreit von Berg und Redi (2023) stellte einen Vergleich zwischen REST (HTTP/1.1 + JSON) und gRPC (HTTP/2 + Protobuf) an und hat dabei den Request Throughput mit einem eigenen Benchmarking-Client gemessen. Es gab vier Klassen von Payloads: XS: 73 B, S: 246 B, M: 4KB, L: 40kB. Die Tests fanden auf einem lokalen Netzwerk statt und die Ergebnisse zeigten: Bei den Klassen XS und S erreichte REST den höheren Durchsatz, bei M und L war gRPC deutlich effizienter und skalierte besser. Begründet wurden die Ergebnisse damit, dass REST weniger Overhead bei

winzigen Nachrichten hat und daher bei sehr kleinen Requests im Vorteil ist. Bei gRPC bringt die binäre Serialisierung und das Multiplexing von HTTP/2 Vorteile sobald mehr Daten übertragen werden und die Anfragefrequenz höher ist. Die Erkenntnis ist daher, dass die Wahl der Technologie stark von der Größe der Payload abhängt (Berg & Redi, 2023).

3.2.4 Performance evaluation of microservices communication with REST, GraphQL, and gRPC

Niswar et al vergleichen die drei API-Protokolle REST, GraphQL und gRPC in einer Microservice-Architektur. In der Arbeit wurden drei Services in Containern mit Redis und MySQL implementiert, die Flat Data und Nested Data abrufen können. Die Performance der Protokolle wird anhand von Responsezeiten und CPU-Auslastungen gemessen. Bei der Messung wurden zwischen 100 bis 500 Requests gesendet, die sowohl parallel als auch sequentiell durchgeführt wurden. Die Ergebnisse zeigen, dass gRPC die schnellsten Antwortzeiten und die geringste CPU-Last aufweist. Erklärt wird dies durch die Verwendung von HTTP/2. REST wies eine mittlere Performance auf, wobei GraphQL die langsamste Antwortzeiten und höchste CPU-Belastung, besonders bei steigender Last, aufwies. Das Fazit des Papers ist daher, dass gRPC am performantesten in der Microservice-Kommunikation ist, gefolgt von REST, welches sinnvoll sein kann wenn Einfachheit im Vordergrund steht. GraphQL ist am ressourcenintensivsten, bietet jedoch Flexibilität (Niswar u. a., 2024).

Kapitel 4

Theoretische Analyse und Vergleich der API-Technologien

Aufbauend auf den theoretischen Grundlagen und dem Stand der Technik, sollen anschließend die ausgewählten API-Technologien: REST, GraphQL, gRPC und gRPC-Web direkt miteinander verglichen werden. Neben den technischen Eigenschaften sollen auch Aspekte wie Erlernbarkeit, Effizienz und Ressourcenverbrauch betrachtet werden. Ziel des Kapitels ist es, die jeweiligen Stärken und Schwächen der Technologien herauszuarbeiten.

4.1 Vor- und Nachteile der API-Technologien

Im folgenden Abschnitt werden die Vor- und Nachteile der API-Architekturen zusammengefasst.

4.1.1 REST

Vorteile:

- REST ist der etablierteste API-Standard für Web-APIs.
- Ein Großteil der Entwickler ist bereits mit REST vertraut.
- Durch die weite Verbreitung gibt es eine Vielzahl an Tools für die Entwicklung und das Testing (z.B. Postman, Swagger) (Postman, 2022, 2023).
- Wird in den meisten Web-Clients nativ unterstützt und liefert typischerweise JSON, welches für Menschen leicht lesbar ist.

- Es ist im Vergleich einfach und intuitiv aufgebaut.
- HTTP-Funktionen wie Caching und Authentifizierung können direkt genutzt werden.
- Jeder Request ist zustandslos, das heißt, alle Kontextinformationen müssen immer mitgeliefert werden. Vorteile davon sind Skalierbarkeit, Ausfallsicherheit und einfacheres Design (Fielding, 2000).

Nachteile:

- Der größte Nachteil von REST ist das Over-Fetching bzw. Under-Fetching. Es gibt klar definierte Endpunkte die einen gewissen Datenansatz zurückliefern. Hierbei kann es zu dem Problem kommen, dass man mehr Daten übertragen muss, als man benötigt, was zu zusätzlichen Netzwerklast und höheren Latenzen führt (Amazon Web Services, 2025).
- Die Zustandslosigkeit hat auch einen Nachteil, alle Kontextinformationen müssen immer mitgeliefert werden, was bei einer Vielzahl von Requests ineffizient ist (Fielding, 2000).
- Versionierung: Änderungen der API erfordern oft neue Endpunktversionen, was die Wartung erschweren kann.
- Auch wenn meist JSON für die Übertragung genutzt wird, gibt es an sich keine strikte Typsicherheit (Red Hat, Inc., 2020).

4.1.2 GraphQL

Vorteile:

- GraphQL bietet eine präzise Datenabfrage, der Client bekommt genau die Menge an Daten, die benötigt wird, Over-Fetching bzw. Under-Fetching wird verhindert.
- Mehrere Ressourcen können in einer Anfrage kombiniert werden, im Gegensatz zu REST, wo mehrere Endpunkte separat aufgerufen werden müssen. Diese Eigenschaft verringert die Netzwerkrundläufe, was vor allem bei langsamen oder mobilen Verbindungen effizient ist.
- GraphQL ist stark typisiert, alle verfügbaren Datentypen und Felder sind definiert und können vom Client abgefragt werden, was die Entwicklung erleichtert.

- Neue Felder können einfach hinzugefügt werden ohne bestehende Queries verändern zu müssen.

Nachteile:

- Die Flexibilität die GraphQL für das Frontend schafft, verlagert die Komplexität auf den Server.
- Die Implementierung kann ohne Batch-Loading oder Caching-Strategien zu Performance Einbußen führen, da meistens alle Abfragen über nur einen einzigen /graphql-Endpoint per POST laufen, und somit das übliche HTTP-Caching-nicht automatisch funktioniert.
- Das fehlen des eingebauten HTTP-Caching führt dazu, dass Entwickler eigeine Cashing-Lösungen implementieren müssen.
- Durch die frei gestaltbaren Queries ist es außerdem schwierig, pauschale Limits oder Vorhersagen über Lasten zu definieren (Amazon Web Services, 2025; GraphQL Foundation, 2025; Red Hat, Inc., 2020).
- Im Gegensatz zu REST ist GraphQL nicht weniger verbreitet und es gibt eine begrenztere Anzahl an Tools (Postman, 2022, 2023).

4.1.3 gRPC

Vorteile:

- gRPC ist für Performance ausgelegt.
- Durch das Verwenden von Protocol Buffers, sind die gesendeten Nachrichten binär kodiert, äußerst kompakt und dadurch schneller übertragen als z.B. JSON. In der Web-Variabte gRPC-Web können diese Vorteile zum Teil auch im Browser genutzt werden.
- In Kombination mit HTTP/2 als Transport ermöglicht gRPC eine sehr niedrige Latenz und effiziente Ausnutzung der Verbindung (Multiplexing).
- gRPC ist streng typisiert, und die zu sendenden Datenstrukturen als auch zur Verfügung stehenden Dienste sind in einer .proto Definition klar definiert.
- Der klare Vertrag zwischen Client und Server erhöht die Typsicherheit, reduziert Missstände und Fehler zur Laufzeit.
- gRPC unterstützt bidirektionales Streaming.

- gRPC generiert automatisch Client-Code für viele Programmiersprachen (gRPC Authors, 2025a).

Nachteile:

- Da gRPC auf HTTP/2 basiert und davon spezielle Features nutzt, die von vielen Webbrowsern nicht unterstützt werden, ist es ohne Weiteres nicht in Browser Umgebungen nutzbar. Die Lösung hierfür ist die Übersetzung von gRPC zu gRPC-Web, welches im Browser benutzt werden kann.
- Die Übersetzung von gRPC zu gRPC-Web erhöht die Infrastruktur-Komplexität, kann zu Fehlerquellen führen, und mindert die Performance (gRPC Authors, 2025b).
- gRPC wird nicht so häufig für die Kommunikation zwischen Web und Backend benutzt, dadurch gibt es weniger Tools und Community Support.
- Die Nutzung von gRPC ist nicht so intuitiv und einfach wie REST und erfordert eine Einarbeitung in neue Tools und Konzepte (Red Hat, Inc., 2020).

4.2 Effizienzvergleich: Latenz, Datenvolumen und Ressourcenverbrauch

Mehrere Studien zeigen, dass gRPC in Bezug auf Latenz, Datenvolumen und CPU-Verbrauch am performantesten ist. Die Ergebnisse werden nicht im Detail wiederholt, sondern in Bezug auf die drei API-Technologien eingeordnet.

Latenz und Antwortzeit: gRPC erreicht durch HTTP/2 und Protobuf geringere Latenzen als REST und GraphQL (Berg & Redi, 2023; Niswar u. a., 2024). REST hat durch das Verwenden von JSON einen Overhead, kann aber bei wiederholten Requests vom Cache profitieren (Red Hat, Inc., 2020). GraphQL weist die höchsten Latenzen auf, kann aber durch individuelle Datenabfrage die Zahl der benötigten Requests minimieren und dadurch in Summe effizienter sein (Amazon Web Services, 2025).

Datenvolumen: gRPC überträgt die Daten durch Protobuf sehr kompakt. REST und GraphQL haben durch die Verwendung vom textbasierten JSON einen Overhead. GraphQL eignet sich am wenigsten für die Übertragung von Blobs, da diese in Base64 kodiert werden müssen, das den Overhead um ein Drittel erhöht (Amazon Web Services, 2025; Berg & Redi, 2023; Red Hat, Inc., 2020).

Ressourcenverbrauch: gRPC ist hinsichtlich der CPU-Auslastung am effizientesten, da Protobuf schneller verarbeitet wird als JSON. REST und GraphQL benötigen durch die aufwendigere Serialisierung und Deserialisierung mehr Rechenressourcen (Berg & Redi, 2023; Niswar u. a., 2024).

4.3 Erlernbarkeit

4.3.1 REST

Abgesehen von der weiten Verbreitung von REST, gilt diese Technologie auch als einfach erlernbar und schnell einsetzbar. Durch die Nutzung von Standard-HTTP Methoden wird eine flache Lernkurve aufgewiesen. Es gibt zahlreiche Tools und Beispiele, was den Einstieg zusätzlich erleichtert. (Postman, 2023; Red Hat, Inc., 2020).

4.3.2 GraphQL

GraphQL bringt aufgrund neuer Konzepte wie, Schemas, Queries, der eigenen Abfragesprache und der Resolver-Logik eine steilere Lernkurve als REST auf (Red Hat, Inc., 2020). Ein kontrolliertes Experiment zeigt jedoch auf, dass die Implementierung konkreter Abfragen mit GraphQL weniger Aufwand erfordern kann als REST (Brito & Valente, 2020). Generell ist GraphQL auch weniger verbreitet als REST wodurch Anfänger auf weniger Ressourcen zurückgreifen können (Postman, 2023).

4.3.3 gRPC

Auch gRPC hat im Vergleich zu REST einen erhöhten Einarbeitungsaufwand, da für die Verwendung die Konzepte von Protocol Buffers, Streaming-Kommunikation und dem Kompilieren von .proto-Dateien vertraut sein müssen. Außerdem braucht man eine spezifische Tools wie protoc und passende Codegeneratoren für die jeweilige Programmiersprache. Entwicklerumfragen zeigen auf, dass gRPC im Vergleich zu REST deutlich seltener eingesetzt wird was aufzeigt, dass Tools, Beispiele und Community Ressourcen weniger verbeitet sind als bei REST (gRPC Authors, 2025a; Red Hat, Inc., 2020).

Kapitel 5

Entwicklung und Umsetzung des Prototyps

Um die Unterschiede bezogen auf Latenz und Performance von REST, GraphQl, gRPC und gRPC-Web in einem konkreten Anwendungsfall zu ermitteln, wurde ein Prototyp entwickelt, mithilfe dessen Messungen zwischen einer Frontend- und Backendanwendung durchgeführt werden können. Bei den Messungen wird die Zeit des Datenaustauschs zwischen den zwei Kommunikationspartnern (End-zu-End Latenz) erfasst und anschließend verglichen und analysiert. Die so gewonnenen Daten sollen anschließend die Basis dafür bilden, um die zuvor theoretisch untersuchten Eigenschaften der Technologien unter realen Bedingungen zu überprüfen.

Zielsetzung

Ziel des praktischen Teils ist es die Leistungsfähigkeit von den genannten API-Technologien zu untersuchen und gegenüberzustellen. Der Prototyp wurde so implementiert, dass für jede Technologie eine vergleichbare Umgebung mit ähnlichen Bedingungen besteht. Im Fokus der Messungen steht die End-zu-End Latenz, also jene Zeitspanne vom Absenden einer Anfrage bis zur vollständigen Verarbeitung und Bereitstellung der Daten im Client.

Ziel der Messungen ist es Unterschiede zwischen den Technologien sichtbar zu machen und mit den im Theorieteil ermittelten Daten zu vergleichen. Auf dieser Grundlage werden anschließend Rückschlüsse auf die Eignung der einzelnen Technologien für die Kommunikation zwischen Frontend- und Backend-Anwendungen gezogen.

5.1 Versuchsaufbau

Der entwickelte Prototyp besteht grundsätzlich aus 3 Komponenten: Ein Backend-Service, ein Web-Client und ein Konsolen-Client.

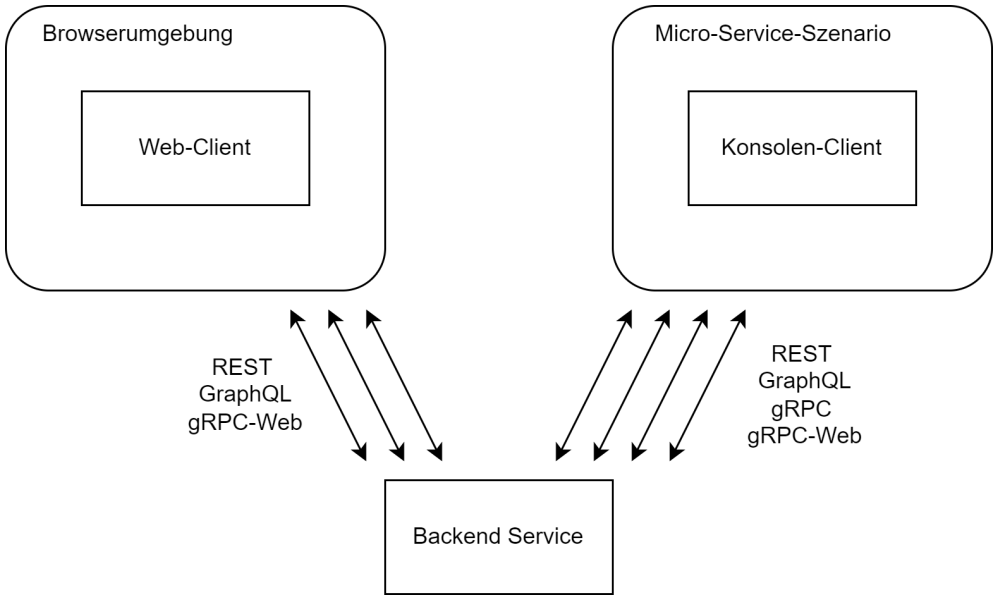

Abbildung 5.1: Schematischer Aufbau des Versuchs

Die zwei verschiedenen Client-Typen können hierbei mit dem gemeinsamen Backend-System kommunizieren.

Der Schwerpunkt der Messungen liegt in folgenden Bereichen:

- Vergleich einzelner und mehrfacher Requests
- Einfluss verschiedener Browser
- Unterschiede zwischen Microservice-Client und Browser-Client

Systemarchitektur:

Front-End Clients:

Der Versuchsaufbau beinhaltet zwei unabhängige Clients, die jeweils auf unterschiedliche Weise auf den Backend-Service zugreifen:

1. Web-Client: die Übertragung findet zwischen einem Browser und dem Back-End Service statt. Der Web-Client kann auf die REST, GraphQL und gRPC-Web API zugreifen.

2. Konsolen Client: die Übertragung findet zwischen einer Microservice ähnlichen Konsolenanwendung und dem Back-End Service statt. Der Konsolen Client kann auf die REST, GraphQL, gRPC und gRPC-Web API zugreifen.

Backend Service:

Der Backend-Service stellt verschiedene Services zur Verfügung und kann von beiden Front-End Clients Anfragen entgegennehmen. Die verschiedenen Services wurden so ausgewählt, dass die gesendeten Daten möglichst realitätsnahe Anwendungsfälle der Frontend-Backend Kommunikation abdecken.

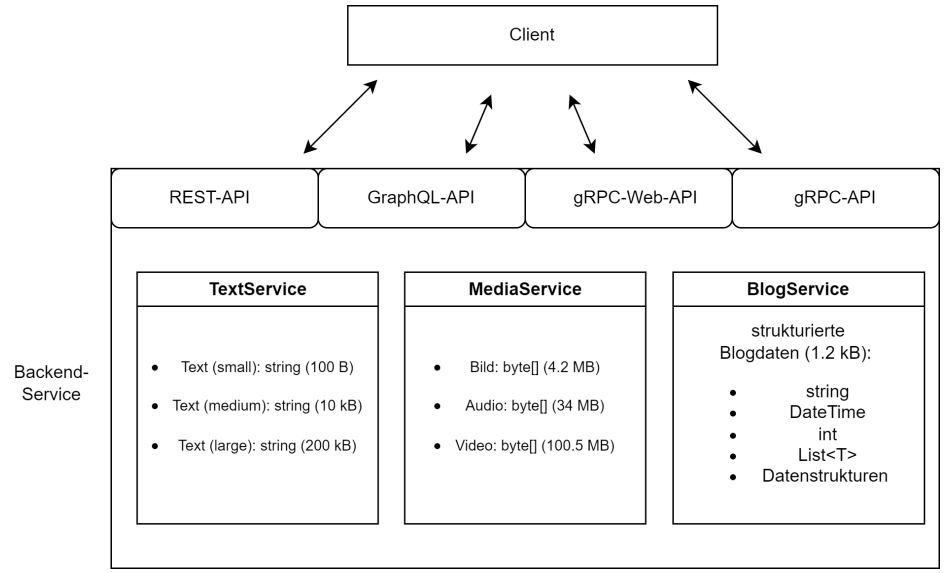

Abbildung 5.2: Schematischer Aufbau des Backend-Services mit Text-, Media- und Blog-Service

Grundsätzlich, kann auf 3 verschiedene Services zugegriffen werden:

- Text-Service: Der Text-Service stellt einen Text im Datentyp „string“ zur Verfügung. Dabei können die Datengrößen 100 B, 10 kB und 200 kB abgefragt werden.
- Media-Service: Der Media-Service stellt Mediendaten zur Verfügung, die mittels eines byte[]-Arrays verschickt werden. Bei den Medien handelt es sich um ein Bild (4.2 MB), einer Audio-Datei (34 MB) und einem Video (100.5 MB).
- Blog-Service: Bei dem Blog-Service handelt es sich um einen 1.2 kB großen Datentypen, der verschachtelt andere Datentypen (int, string, DateTime, Listen, Klassen) bereitstellt.

Alle Services wurden separat mit den Technologien REST, GraphQL, gRPC und gRPC-

Web implementiert und können somit von den jeweiligen Clients abgefragt und verglichen werden.

Testfälle:

Zur Analyse der Client-Server Kommunikation wurden folgende Testfälle definiert:

- **Einzelabfragen - Web-Client (1 Request):** Vergleich der Technologien bei beim senden eines einzelnen Requests. Dies wird jeweils bei jedem der definierten Services und für jede vorhandene Datengröße durchgeführt. Die Abfragen werden jeweils mit REST, GraphQL und gRPC-Web durchgeführt. Die Messungen zeigen Ergebnisse über Latenz und Performance.
- **Mehrfachabfragen - Web-Client (20 Requests):** Analog zum Einzelrequest, wird der Versuch erneut mit parallelen Mehrfachabfragen durchgeführt, um zu sehen, wie gut die Technologien skalieren, Ressourcen nutzen und mehrere Requests effizient handhaben können.
- **Unterschiede bei der Nutzung verschiedener Browser:** Um den Einfluss des Webbrowsers auf die Messungen des Web-Clients zu identifizieren, werden erneut Einzelrequests in verschiedenen Webbrowsern durchgeführt. Durch den Versuch wird festgestellt, ob eine Abhängigkeit des Browsers auf die Technologie besteht.
- **Einzelabfragen - Konsolen-Client (1 Request):** Analog zu den Einzelrequests des Web-Clients wird der Versuch erneut mit einem mircoservice-ähnlichen Konsolenclient durchgeführt. Hierbei wird jedoch auch abgesehen von REST, GraphQL und gRPC-Web die Latenz von gRPC gemessen. Durch den Versuch ist eine Gegenüberstellung der Response Zeiten zwischen den beiden Clients möglich.

Da es in Web Browsern nicht möglich ist normales"gRPC zu verwenden, konnte diese Messung im Rahmen des Browser-Client-Versuchsaufbaus nicht durchgeführt werden. Um Messwerte zwischen gRPC und gRPC-Web zu vergleichen, und um zu sehen inwiefern sich die Performance zwischen Browser-Backend und Microservice-Backend unterscheiden, wurde der zweite Versuchsaufbau mittels eines Konsolen-Clients durchgeführt.

Rahmenbedingungen

Um einen fairen Vergleich herzustellen, wurden alle Requests ausschließlich mit dem HTTP/2 Protokoll durchgeführt und verschlüsselt mit HTTPS versendet. Bei der Response-Zeit handelt es sich jeweils um die End-zu-End-Latenz aus der Sicht des Nutzers. Es wird dabei die Zeit ab dem Absenden des Requestes bis zu dem Zeitpunkt, ab dem die Daten tatsächlichen im Client fertig verarbeitet bereitstehen gemessen. Die Response Zeit beinhaltet also:

1. die Transportzeit über das Netzwerk: Zeit für den Hin- und Rückweg des Pakets durch das Netzwerk.
2. Serverseitige-Verarbeitung: Die serverseitige Verarbeitung des Requests
3. Antwortübertragung: Zeit für das Senden der Antwort(inklusive Header und Payload) zurück an den Client.
4. Clientseitige Verarbeitung: die Verarbeitung des Responses, sodass die Daten tatsächlich im Client verwendet werden können. (JSON Parsing, Binär-Parsing).

5.2 Implementierung

Der Prototyp beinhaltet einen Backend-Service, einen Web-Client und einen Konsolen Client. Dieser Abschnitt befasst sich mit der Implementierung der jeweiligen Komponenten.

Um die Vergleichbarkeit der jeweiligen API-Technologien zu erhöhen, wurden alle Clients und Services so minimalistisch wie möglich umgesetzt, da weitere Frameworks oder Logik die Datenverarbeitung beeinflussen, und somit die tatsächlichen Zeiten verzerren könnten. Jeder Service liefert ausschließlich die für den Testfall angeforderten Daten. Die konkrete Implementierung ist auf GitHub verfügbar (siehe Quelle: Mühlberger, 2025).

Ziel dieses Abschnittes ist es die Architekturentscheidungen, Funktionsweisen sowie die technischen Aspekte darzustellen, auf Basis anschließend die Messungen durchgeführt werden.

5.2.1 Backend-Service:

Der Backend Service wurde modular und technisch getrennt umgesetzt, um die Technologien jeweils unabhängig voneinander vergleichen zu können. Die Implementierung basiert auf .NET 9, einer plattformunabhängigen Open-Source-Laufzeitumgebung von Microsoft, die für moderne Web- und Microservice-Anwendungen konzipiert ist. Die Services stellen für jede der APIs dieselben Daten zur Verfügung und können folgendermaßen abgefragt werden:

- REST: per HTTP-GET auf definierte Endpunkte.
- GraphQL: über einen einzigen HTTP-POST Endpunkt, wobei der Client die benötigten Felder in der Query spezifiziert.
- gRPC und gRPC-Web: als typisierte RPC calls (Protobuf)

Tabelle 5.1: Vergleich der API-Endpunkte

Inhalt	**REST-Endpunkt**	**gRPC-Methode**	**GraphQL-Abfrage**
Großer Text	`GET /text/large`	`Text.GetLarge()`	`query { large { content } }`
Video (MP4)	`GET /media/video`	`Media.GetVideo()`	`query { video }`
Blogposts	`GET /api/blog`	`Blog.GetAll()`	`query { posts { title, author... } }`

Die Struktur des Backend-Services gliedert sich in 4 eigenständige Projekte:

1. Common: Das Common Projekt stellt die Testdaten zentral zur Verfügung. Damit die Testdaten bei einem Request nicht zuerst aus der Datei gelesen werden müssen, werden diese bei Programmstart einmal in den Arbeitsspeicher geladen. Zu diesem Zweck wurde ein API-Cache erstellt, der alle Text und Medienobjekte vorab einliest. Die Daten können dann zur Laufzeit direkt vom Cache abgefragt werden.

   Für die Testdaten wurden Datentypen die typischerweise an Web Clients gesendet werden ausgewählt und beinhalten:

   - Texte in drei Größen: 100 B, 10 kB, 200 kB
   - Medieninhalte: 4.2 MB Foto, 34 MB Audio-Datei, 100.5 MB Video-Datei
   - Blogdaten: ein selbst definierter Datentyp mit verschachtelten Abschnitten, Metadaten, Zahlenblöcken und Autoreninformationen - 1.2 kB

2. RestAPI: Die REST API wurde mittels einem Controller Muster von ASP.NET Core implementiert und stellt 3 eigenständige Endpunkte zur Verfügung. Die Kommunikation erfolgt ausschließlich über HTTP GET.

   Die Controller sind wie folgt definiert:

   - **TextController**: Stellt Textdaten in drei Größen zur Verfügung.
     Rückgabeformat: `application/json`
   - **MediaController**: Stellt ein Bild, eine Audiodatei und ein Video bereit.
     Rückgabeformat: Binärdaten (`byte[]`) mit MIME-Type `image/jpeg`, `audio/wav`, `video/mp4`
   - **BlogController**: Stellt einen vordefinierten Blogeintrag bereit.
     Rückgabeformat: `application/json`

3. GraphQLAPI: Die GraphQL-API wurde mittels des `.NET`-Frameworks `HotChocolate` implementiert. Als Einstiegspunkt wurde eine `Query`-Klasse definiert, welche für die verschiedenen Services (Text, Medien, Blog) jeweils erweitert wurde.

   Folgende Queries wurden definiert:

   - **TextQuery**: Stellt die Felder `small`, `medium` und `large` bereit, welche jeweils Textinhalte als `string` zurückgeben.
   - **MediaQuery**: Stellt die Felder `image`, `audio` und `video` bereit, welche in der GraphQL-Antwort Base64-kodiert übertragen werden.

- **BlogQuery**: Stellt das Feld `posts` zur Verfügung, welches die für den Blogpost definierten Daten enthält.

Alle Abfragen erfolgen über `HTTP POST`-Anfragen an den Endpunkt `/graphql` und haben das Format `application/json`.

4. GrpcWebAPI: Die gRPC-Web-API basiert auf den definierten `.proto`-Dateien (`text.proto`, `media.proto`, `blog.proto`). In den Proto-Dateien wurden sowohl die *Messages* und Datentypen der Testdaten als auch die *Services*, mit denen auf die Messages zugegriffen werden kann, definiert.

   Die Services implementieren folgende Methoden:

   - **TextService**: Rückgabe der drei Textgrößen über die Methoden `GetSmall()`, `GetMedium()` und `GetLarge()`.
     Rückgabeformat: `TextResponse`-Message mit einem `string`-Feld (`content`), serialisiert im Protocol-Buffers-Format.

   - **MediaService**: Streaming von Binärdaten in 64 kB-Chunks über die Methoden `GetImage()`, `GetAudio()` und `GetVideo()`. Anders als bei den anderen Services werden hier die Mediendateien nicht als vollständige Datei, sondern als 64 kB-Chunks gestreamt. Da gRPC nicht dafür konzipiert wurde, große Dateien am Stück zu übertragen, ist dies die empfohlene Vorgehensweise für den Umgang mit großen Dateien. Bei REST und GraphQL hingegen wurde auf ein solches Streaming verzichtet, da entsprechende Mechanismen nicht nativ unterstützt werden.
     Rückgabeformat: Server-Streaming von Chunk-Nachrichten (`bytes`-Feld), serialisiert als Protocol-Buffers-Stream über HTTP/2.

   - **BlogService**: Rückgabe strukturierter Blogposts via `GetAll()`.
     Rückgabeformat: `BlogPostsResponse` (Liste von `BlogPost`-Nachrichten), ebenfalls als Protocol Buffers kodiert.

   Die API wurde so konfiguriert, dass sowohl normales gRPC als auch gRPC-Web verwendet werden kann. Die Implementierung für gRPC-Web wird mittels des Pakets `Grpc.AspNetCore.Web` bereitgestellt. Die Generierung der gRPC-eigenen Klassen erfolgte mittels *gRPC Tools*. *gRPC Tools* erstellt dabei bei jedem Build die im Proto-File definierten Klassen, die anschließend im Code verwendet werden können.

   Tabelle 5.2 zeigt Libraries, Frameworks und Tools, sowie deren Versionen welche für die Implementierung des Backend-Services eingesetzt wurden.

Tabelle 5.2: Verwendete Technologien Backend

Komponente	Technologie/Tool	Version
Backend-Framework	`.NET SDK`	9.0
REST API	`ASP.NET Core Web API`	9.0.5
GraphQL	`HotChocolate`	15.1.5
gRPC	`Grpc.AspNetCore`	2.71.0
gRPC-Web	`Grpc.AspNetCore.Web`	2.71.0
Protobuf Tools	`Grpc.Tools` / `protoc`	2.72.0

5.2.2 Web-Client:

Um die Messungen aus der Sicht eines Front-End-Clients durchzuführen, wurde ein Web-Client erstellt. Von der Anwendung aus, können die Requests an die jeweiligen Services gesendet werden. Beim Senden eines Requests wird die End-zu-End Latenz im Client gemessen und anschließend mit der Payload grafisch angezeigt.

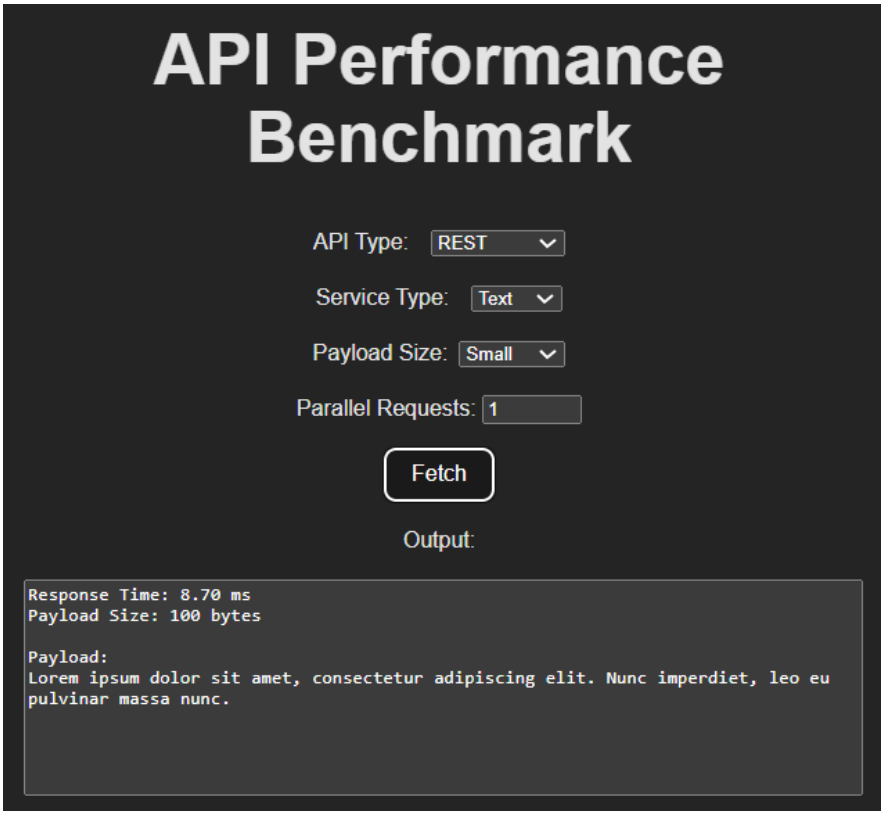

Abbildung 5.3: Interface - Web-Client

Die Implementierung wurde mit `React` und `TypeScript` durchgeführt und mithilfe von `Vite` gebündelt und lokal bereitgestellt.

Architektur und Aufbau

Der Web-Client dient primär als Simulation eines realen Benutzerverhaltens einer Front-End-Anwendung. Über ein einfaches Interface kann ausgewählt werden:

- Gegen welche API-Schnittstelle (`REST`, `GraphQL`, `gRPC-Web`) ein Request ausgeführt werden soll
- Welche Datenart (Text, Medien, Blog) und welche Datenmenge übertragen werden soll
- Ob ein einzelner Request oder eine Mehrfachanfrage parallel durchgeführt werden soll

Sowohl bei REST als auch bei GraphQL wird im Frontend die browsernative `fetch`-API für die Kommunikation mit dem Backend verwendet. Für die Kommunikation mit der gRPC-Web-Schnittstelle werden gRPC-Bibliotheken eingesetzt. Die Generierung der gRPC-Klassen erfolgt mittels der `ts-protoc-gen`-Bibliothek mit folgendem Befehl:

```
npx protoc --ts_out src/api/generated
 --proto_path=src/proto src/proto/*.proto
```

Clientseitige Datenverarbeitung

Nach erfolgreichem Empfang der Daten vom Server müssen die Daten vom Client verarbeitet werden, damit diese anschließend verwendet werden können. Abhängig vom ausgewählten API-Typ und der Datenmenge werden die Daten jeweils unterschiedlich verarbeitet.

Textdaten

- Bei REST und GraphQL erfolgt das Parsen des gesendeten JSON mittels der nativen `fetch`-API.
- Bei gRPC-Web wird die Protobuf-Nachricht automatisch über die generierten TypeScript-Klassen des `protobuf-ts`-Plugins deserialisiert.

Mediendaten Medieninhalte werden binär als `byte[]-Array` übertragen.

- Bei REST erfolgt der Empfang direkt als Blob über `response.blob()`.

- Bei GraphQL werden die Binärdaten Base64-kodiert als String im JSON-Response übertragen und im Client manuell dekodiert.
- Bei gRPC-Web werden die Binärdaten als Daten-Chunks gestreamt. Die Chunks werden im Client rekonstruiert.

Blogdaten

- Bei REST und GraphQL werden JSON-Objekte mit verschachtelter Struktur geparsed.
- Bei gRPC-Web erfolgt die Deserialisierung über generierte Protobuf-Klassen.

Tabelle 5.3 zeigt die Libraries, Frameworks und Tools, sowie deren Versionen welche für die Implementierung des Web-Clients eingesetzt wurden.

Tabelle 5.3: Verwendete Frontend-Technologien des Web-Clients

Komponente	Technologie/Tool	Version
Frontend-Framework	`React`	19.1.0
Bundler	`Vite`	6.3.5
TypeScript Compiler	`TypeScript`	5.8.3
REST	`fetch API` (Browser native)	(native)
GraphQL Client	`fetch API` (Browser native)	(native)
gRPC-Web Transport	`grpcweb-transport`	2.11.0
Protobuf TS Plugin	`protobuf-ts`	2.11.0
Protoc Codegen Plugin	`ts-protoc-gen`	0.15.0
gRPC-Web Codegen	`protoc-gen-grpc-web`	1.5.0

5.2.3 Konsolen-Client

Um Messungen mit gRPC durchführen zu können wurde ein Konsolen-Clients implementiert. Dadurch können Response-Zeiten zwischen gRPC und gRPC-Web direkt miteinander verglichen werden.

Der Konsolen-Client kann alle unterstützten APIs (`REST`, `GraphQL`, `gRPC` und `gRPC-Web`) ansprechen und führt Messungen über ein textbasiertes Menüsystem durch. Es werden nur Einzel-Requests unterstützt.

Kommunikationsmechanismen

Die Kommunikation erfolgt jeweils über folgende Mechanismen:

- **REST**: per `HttpClient` mit JSON-Deserialisierung

- **GraphQL**: per `GraphQL.Client`-Bibliothek
- **gRPC**: über `Grpc.Net.Client` direkt als native gRPC-Kommunikation
- **gRPC-Web**: über `Grpc.Net.Client.Web` mittels spezieller Web-Handler, angepasst an das gRPC-Web-Protokoll

Die interne Verarbeitung wurde so implementiert, dass die Messungen die Response-Zeiten und anschließend das Parsen beinhalten. Bei den Mediendaten findet auf Client-Seite keine weitere Verarbeitung des `byte[]`-Arrays statt, daher ist hier eine Gegenüberstellung zum Web-Client nicht sinnvoll. Tabelle 5.4 zeigt die Libraries, Frameworks und Tools, sowie deren Versionen welche für die Implementierung des Konsolen-Clients eingesetzt wurden.

Tabelle 5.4: Verwendete Technologien des Konsolen-Clients

Komponente	Technologie/Tool	Version
Client-Framework	`.NET SDK`	9.0
GraphQL Client	`GraphQL.Client`	6.1.0
GraphQL Client Serializer	`GraphQL.Client.Serializer.Newtonsoft`	6.1.0
gRPC Client	`Grpc.Net.Client`	2.71.0
gRPC-Web Client	`Grpc.Net.Client.Web`	2.71.0
gRPC Code Generator	`Grpc.Tools`	2.72.0

5.3 Spezifikationen des Testsystems

Um die Messergebnisse korrekt und vergleichbar einordnen zu können, ist es notwendig darzustellen, unter welchen technischen Rahmenbedingungen die jeweiligen Messungen durchgeführt wurden. Da die Leistungsfähigkeit der Hardware sowie die Softwareversionen die Messzeiten beeinflussen können. Für die Messungen wurde folgendes Testgerät verwendet: Lenovo ThinkPad X1 Carbon der 10. Generation (Modellbezeichnung: 21CB)

Hardwarekonfiguration

- **Prozessor (CPU)**: 12th Gen Intel® Core™ i5-1245U, 1600 MHz, 10 Kerne
- **Arbeitsspeicher (RAM)**: 16 GB LPDDR5, 5200 MT/s
- **Massenspeicher (SSD)**: 512 GB NVMe SSD

Betriebssystem: Microsoft Windows 11 Pro (Version: 10.0.26100, Build 26100)

5.4 Messung

Ziel der Messungen ist es, die End-zu-End-Latenz und Performance bei der Datenübertragung zwischen Frontend und Backend zu quantifizieren und somit die im theoretischen Teil beschriebenen Eigenschaften der verschiedenen API-Technologien mit einem praktischem Anwendungsfall zu vergleichen. Die Responsezeiten beschreiben die tatsächliche Zeit, bis die jeweiligen Daten im Client bereitstehen (Datenübertragung und Parsing). Dabei wurden unterschiedliche Datenarten verwendet, die häufig in der Frontend-zu-Backend-Kommunikation zum Einsatz kommen.

Untersucht wurde die Response-Zeit von Einzel-Requests, Mehrfachabfragen (20 parallele Requests) und browserabhängige Unterschiede in einem Web-Client. Zusätzlich wurden alle Technologien (REST, GraphQL, gRPC-Web und gRPC) im Konsolen-Client getestet, um Unterschiede zwischen browserbasierter und microservice-orientierter Kommunikation sichtbar zu machen und Unterschiede zwischen gRPC und gRPC-Web zu ermitteln.

Für den Vergleich der Messwerte wurde der gerundete arithmetische Mittelwert herangezogen, sofern die Abweichung zum Median nicht zu groß war und damit ausgeschlossen werden konnte, dass der Mittelwert durch Ausreißer verzerrt wurde. Zur besseren grafischen Darstellung der Daten wurde die y-Achse der Diagramme logarithmisch skaliert.

Die Messungen wurden ausschließlich lokal auf localhost durchgeführt. Dadurch können zusätzliche Einflüsse wie Paketverlust, Routing-Latenz oder Bandbreitenschwankungen, die bei realen Bedingungen auftreten würden, ausgeschlossen worden. Ziel der Messungen war es, tatsächliche Unterschiede zwischen den API Technologien (gRPC, gRPC-Web, REST, GraphQL) unter optimalen Bedingungen und reproduzierbar zu ermitteln. Der Fokus wurde ausschließlich auf den Vergleich der API-Architekturen und deren Verarbeitung gelegt.

Die Ergebnisse der Messungen liefern eine Grundlage zur Beantwortung der Forschungsfragen, inwiefern sich die Schnittstellentechnologien hinsichtlich der Effizienz und Eignung für typische Frontend-Szenarien unterscheiden. Die nachfolgenden Abschnitte stellen die gemessenen Daten dar und geben einen direkten Vergleich der Technologien.

5.4.1 Web-Client (Einzel-Request)

Bei der Messung der End-zu-End Latenz ergaben sich zwischen den einzelnen Technologien erhebliche Unterschiede. Für eine sinnvolle Auswertung der Daten, wurden jeweils 30 Requests unabhängig voneinander durchgeführt und die Durschnittszeiten und der Median aus den gemessenen Daten ermittelt.

Tabelle 5.5: Chrome-Client – Einzel-Requests: Durchschnitt und Median (Antwortzeit in ms)

Technologie	**Service**	**Datentyp**	**AVG**	**MEDIAN**
REST	Text	small text	8.6	8.7
		medium text	9.0	8.8
		large text	13.2	12.9
	Media	Foto	36.1	36.5
		Audio	148.9	151.6
		Video	365.0	353.6
	Blog	-	8.6	8.2
GraphQL	Text	small text	7.6	7.4
		medium text	7.7	7.6
		large text	10.1	9.8
	Media	Foto	511.7	490.5
		Audio	5896.9	5658.3
		Video	-	-
	Blog	-	7.6	7.2
gRPC-Web	Text	small text	7.5	7.5
		medium text	8.4	8.3
		large text	18.8	18.8
	Media	Foto	182.0	162.6
		Audio	2372.5	2355.0
		Video	-	-
	Blog	-	7.9	7.8

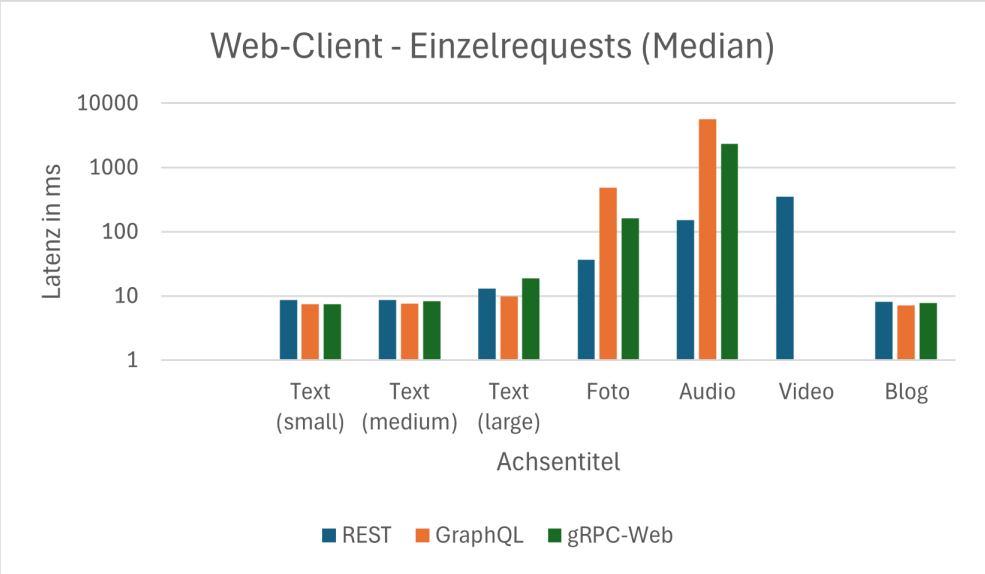

Abbildung 5.4: Web-Client - Einzel-Request: End-zu-End-Latenz (Median) je Datentyp und Technologie [ms]

Text-Service: Es ist zu erkennen, dass kleinere Textdaten (100 B-10 kB) von allen Schnittstellen relativ schnell verarbeitet werden konnten. gRPC-Web und GraphQL weisen hierbei jedoch die geringsten Antwortzeiten (7.5-8.5 ms) auf, REST benötigte ungefähr 8-9 ms für kleinere Textdaten. Bei größeren Textmengen (200 kB) steigt die Latenz erwartungsgemäß bei allen Technologien etwas an, wobei diese bei GraphQL mit 10 ms und REST mit 13 ms deutlich schneller als bei gRPC-Web (19 ms) bereitstehen.

Media-Service: Besonders auffällig ist der Leistungsunterschied bei der Übertragung von Mediendaten. Hierbei war entgegen der theortischen Erwartungen REST die schnellste Technologie für alle Medienarten. Für Bilder (4MB) lag die mittlere Antwortzeit bei 36 ms, deutlich vor gRPC (182 ms) und GraqhQL (511 ms). Ähnliches Verhalten zeigte sich bei der Übertragung von Audiodateien (34 MB), hierbei wurde mit REST eine mittlere Antwortzeit von 149 ms gemessen, mit deutlichem Abstand gefolgt von gRPC-Web (2372 mms) und GraphQL (5897 ms). Bei der Übertagung von Videodaten (100 MB) konnte in Chrome keine vergleichbare Messung der Antwortzeiten durchgeführt werden. Die Übertragung war nur über die REST-API erfolgreich. Es ergab sich dabei eine durchschnittliche Antwortzeit von 365 ms. Wegen großer Ladezeit bzw. Instabilitäten des Clients konnten für GraphQL und gRPC-Web keine Daten gemessen werden.

Blog-Service: Bei der Messung der Blogdaten, konnten geringe Unterschiede in den Abfragezeiten zwischen den Technologien identifiziert werden. GraphQL war hierbei

mit einer mittleren Latenz von 7.6 ms leicht schneller als gRPC-Web (7.9 ms) und REST (8.6 ms).

5.4.2 Web-Client (Parallele-Requests)

Bei der Durchführung von 20 parallelen Requests wurden jeweils pro API und Service 30 Messwerte (Ausnahme: Media-Service) gemessen und anschließend der Durchschnittswert und Median berechnet. Es zeigten sich erwartungsgemäß höhere Latenzen gegenüber der Einzelabfragen.

Tabelle 5.6: Chrome-Client – 20 parallele Requests: Durchschnitt und Median (Antwortzeit in ms)

Technologie	**Service**	**Datentyp**	**AVG**	**Median**
REST	Text	small text	45.5	44.0
		medium text	61.9	62.0
		large text	92.1	93.6
	Media	Foto	385.8	384.6
		Audio	2420.1	2393.4
		Video	-	-
	Blog	-	53.0	48.3
GraphQL	Text	small text	31.4	29.0
		medium text	35.4	33.4
		large text	71.1	68.7
	Media	Foto	16159.6	15745.7
		Audio	-	-
		Video	-	-
	Blog	-	34.6	33.0
gRPC-Web	Text	small text	33.4	32.1
		medium text	43.5	40.9
		large text	145.3	145.4
	Media	Foto	5736.2	5682.4
		Audio	126026.5	107938.0
		Video	-	-
	Blog	-	34.5	31.3

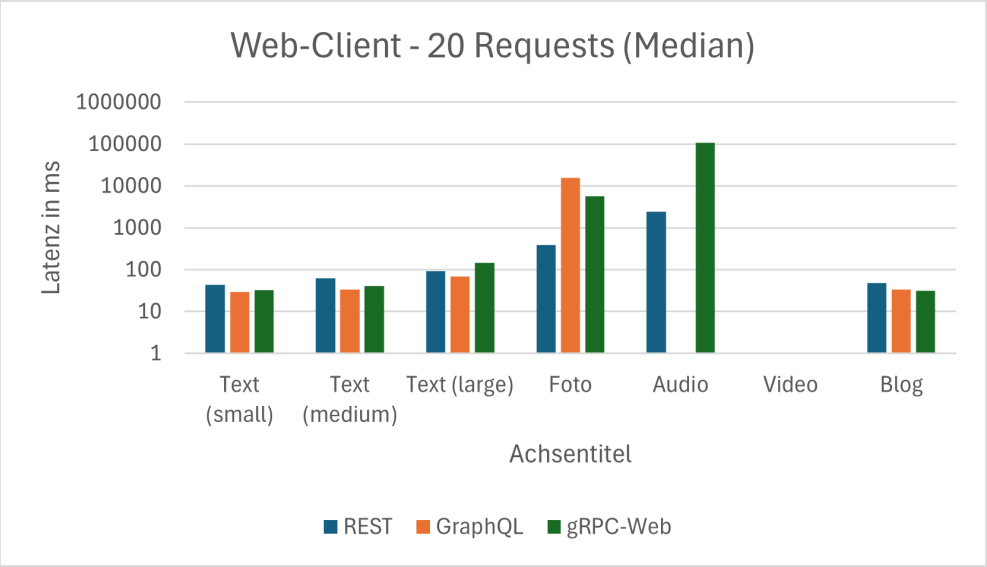

Abbildung 5.5: Web-Client - 20 parallele Requests: End-zu-End-Latenz (Median) je Datentyp und Technologie [ms]

Text-Service: Während kleinere Textdaten noch performant verarbeitet werden können, nehmen die Unterschiede bei größeren Lasten stärker zu. Bei sehr kleinen Textdaten von 100 B sind gRPC-Web und GraphQL jeweils mit einer Antwortzeit von 31ms (GraphQL) und 33 ms (gRPC-Web) gleichauf, bei 10 kB ist GraphQL mit 35 ms ungefähr 9 ms schneller als gRPC-Web mit 44 ms. Je größer die Daten werden, desto langsamer wird gRPC-Web. REST benötigte bei den kleineren Lasten 46 ms für 100 B und 62 ms für 10 kB. Auch bei größeren Textdaten ist GraphQL mit 71 ms am schnellsten, gefolgt von REST mit 92 ms und gRPC-Web mit 145 ms.

Media-Service: Bei der Verarbeitung von Medien kam es wie bereits durch die einzelabfragen erwartet zu größeren Unterschieden. REST schnitt auch hier mit 386 ms für Fotos und 2420 ms für Audiodaten am besten ab. Gefolgt von gRPC-Web mit 5736 ms für das Foto und 126026 ms für die Audiodatei. Bei GraphQL dauerte der Request für die Fotodaten 16159 ms. Für Video- bzw. Audiodateien (GraphQL) wurden aufgrund des bereits bei den Einzelabfragen instabilen Ladeverhaltens keine weiteren Messungen durchgeführt.

Blog-Service: Bei den Abfragen der Blogdaten lagen die Antwortzeiten aller Technologien im zweistelligen ms Bereich. Ähnlich wie bei kleineren Textdaten, sind gRPC-Web und GraphQL mit jeweils 35 ms gleichauf, gefolgt von REST mit 53 ms.

5.4.3 Browser-Vergleich

Derselbe Versuch mit Einzel-Requests wurde anschließend in zwei weiteren Web Browsern (Firefox und Microsoft Edge) durchgeführt, um festzustellen, ob es bei der Performance der APIs eine Abhängigkeit vom Webbrowser gibt. Bei der Messung wurden jeweils 10 Requests pro Technologie und Service durchgeführt und daraus der Durchschnitt und der Median ermittelt.

Tabelle 5.7: Vergleich WebClient (1 Request) – Chrome, Firefox, Edge (AVG in ms)

Technologie	**Service**	**Datentyp**	**Chrome**	**Firefox**	**Edge**
REST	Text	small text	8.6	8.5	4.84
		medium text	9.0	9.3	6.31
		large text	13.2	11.4	9.5
	Media	Foto	36.1	54.4	36.3
		Audio	148.9	248.7	144.9
		Video	365.0	676.4	422.8
	Blog	-	8.6	6.7	7.9
GraphQL	Text	small text	7.6	9.4	5.61
		medium text	7.7	11.1	5.68
		large text	10.1	15.8	8.21
	Media	Foto	511.7	1743.4	402.8
		Audio	5896.9	13253.3	6442.1
		Video	-	90091	-
	Blog	-	7.6	9.6	5.6
gRPC-Web	Text	small text	7.5	9.4	5.5
		medium text	8.4	9.4	5.7
		large text	18.8	24.2	20.3
	Media	Foto	182.0	248.1	186.4
		Audio	2372.5	2455.6	2744.9
		Video	-	8984.8	7984.6
	Blog	-	7.9	8.0	5.2

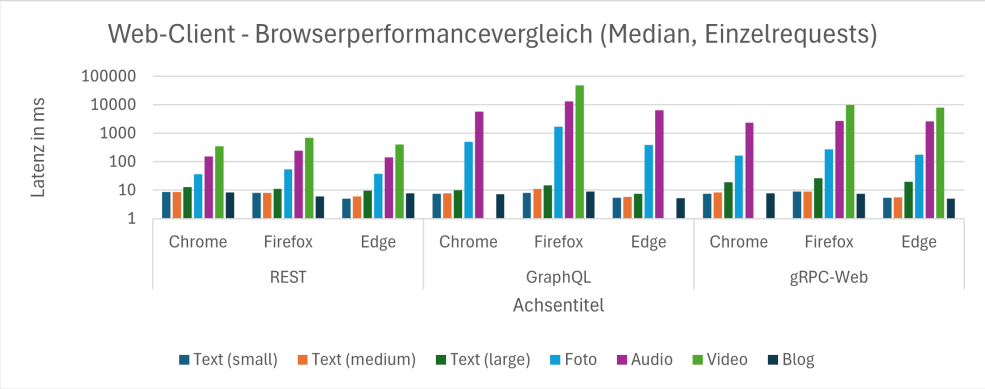

Abbildung 5.6: Web-Client - Browser-Vergleich (Einzel-Request): End-zu-End-Latenz (Median) je Datentyp und Technologie [ms]

Textdaten: Bei den Messungen reagierte Edge in allen Schnittstellen mit den niedrigsten Antwortzeiten, dabei konnten bei allen drei API-Architekturen deutliche Unterschiede gemessen werden. Für kleinere Textdaten (100 B bis 10 kB) benötigten Chrome und Firefox bei REST 8.5-9.5 ms während Edge nur 4.8 und 6.3 ms benötigte. Ähnlich ist das Verhalten bei GraphQL und gRPC-Web wobei Chrome bei diesen Architekturen deutlich schneller als Firefox ist. Auch bei größeren Textmengen blieb Edge bei allen Technologien am performantesten. Bei GraphQL und gRPC-Web war Chrome wieder schneller als Firefox. Bei dem REST Service war Firefox jedoch mit 11 ms schneller als Chrome (13 ms).

Medieninhalte (Foto, Audio, Video) Beim Senden von Mediendaten hatten Chrome und Edge im Gegensatz zu Firefox eine ähnliche Performance. Bei REST und gRPC-Web konnten Edge und Chrome die kürzesten Antwortzeiten für Fotos (36 ms) und Audiodateien (ungefähr 150 ms) liefern. Bei Firefox lagen diese bei REST bei 54 ms und 249 ms bzw. bei gRPC-Web bei 250 ms und 2455 ms.

Bei dem Senden von Videodaten war Chrome am performantesten. Hier hatte REST eine Antwortzeit von 365 ms, Edge 423 ms und Firefox 676 ms. Im Gegensatz zu Chrome, konnte das Video in Firefox mit gRPC-Web (8984 ms) und Edge (7984 ms) dargestellt werden. In Firefox war sogar ein laden mit GraphQL möglich (90091 ms), auch wenn dies sehr lange dauerte.

Blogdaten: Wie bei Textdaten ist Edge auch bei dem Senden von Blogdaten in fast allen Kategorien am schnellsten. Während Chrome und Firefox bei gRPC-Web mit 8 ms gleich schnell sind, ist Firefox bei REST mit 7 ms etwas schneller als Chrome (9ms). Bei GraphQL ist wieder Chrome mit 8 ms schneller als Firefox (10 ms).

5.4.4 Konsolen-Client (Einzel-Request)

Um die Kommunikationsleistung der einzelnen Technologien ohne Einfluss eines Browers aufzuzeigen, und um einen direkten Performancevergleich zwischen gRPC und gRPC-Web zu ermitteln wurden ebenfalls Messungen in der Kommunikation innerhalb einer Microservice basierten Architektur untersucht. Wie erwartet sind die Messwerte in dieser Messung zum größten Teil deutlich besser als zwischen Web-Client und Backend Service.

Tabelle 5.8: Konsole – 1 Request: Durchschnitt und Median (Antwortzeit in ms)

Technologie	**Service**	**Datentyp**	**AVG**	**Median**
REST	Text	small text	2.3	2.0
		medium text	2.1	2.0
		large text	6.4	7.0
	Media	Foto	45.9	41.5
		Audio	250.4	249.5
		Video	540.2	515.0
	Blog	-	1.0	1.0
GraphQL	Text	small text	2.3	2.0
		medium text	2.2	2.0
		large text	5.5	5.0
	Media	Foto	5716.2	5742.0
		Audio	-	-
		Video	-	-
	Blog	-	2.4	2.0
gRPC-Web	Text	small text	2.2	2.0
		medium text	2.8	3.0
		large text	6.8	7.0
	Media	Foto	51.3	53.0
		Audio	251.4	248.0
		Video	705.1	717.5
	Blog	-	2.9	3.0
gRPC	Text	small text	1.2	1.0
		medium text	1.5	1.5
		large text	4.1	4.0
	Media	Foto	42.7	41.0
		Audio	252.1	249.0
		Video	610.0	615.0
	Blog	-	1.1	1.0

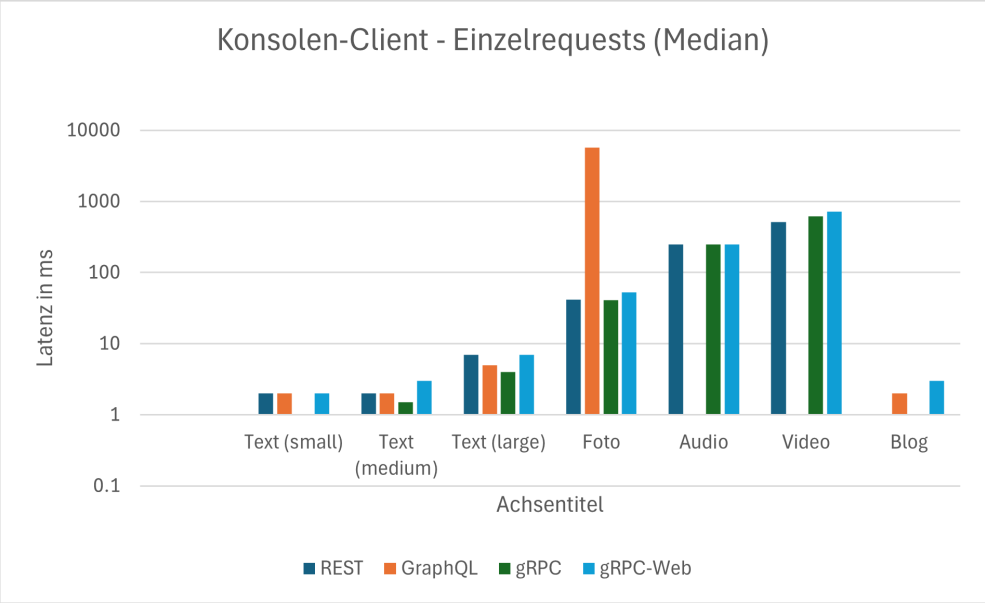

Abbildung 5.7: Konsolen-Client - Einzel-Request: End-zu-End-Latenz (Median) je Datentyp und Technologie [ms]

Text-Service: Alle Technologien zeigten bei kleinen und mittleren Textdaten im Konsolen-Client sehr geringe Antwortzeiten. gRPC hatte hier jedoch eindeutig die schnellste Response Zeit mit nur 1.2 ms für 100 B und 1.5 ms für 10 kB. REST, GraphQL und gRPC-Web haben ungefähr die selbe Reponse-Zeit von 2 ms für beide Textgrößen wobei gRPC-Web mit 2.2 ms für 100 B und 2.8 ms für 10 kB am langsamsten ist. Auch bei großen Textdaten ist gRPC mit 4.1 ms am schnellste, gefolgt von GraphQL mit 5.5 ms, REST mit 6.4 ms und gRPC-Web mit 6.8 ms.

Media-Service: gRPC hatte mit 43 ms die schnellste Übertragung von Fotos gefolgt von REST mit 46 ms. gRPC-Web war mit 51 ms etwas langsamer und GraphQL mit 5716 ms deutlich am langsamsten. Bei den 30 MB Audiodaten sind REST, gRPC und gRPC-Web mit ungefähr 250 ms gleichauf. Nur bei großen Binärdaten ist REST mit 540 ms schneller als gRPC (610) und gRPC-Web (705 ms). Für GraphQL konnte keine sinnvolle Messung für Audio oder Videodaten durchgeführt werden.

Blog-Service: Bei dem Senden von strukturierten Bloddaten liegen gRPC und REST mit ungefähr 1 ms ungefähr gleich auf, gefolgt von GraphQL mit 2,4 ms und gRPC-Web mit 2.9 ms. Auch hier werden die Daten deutlich schneller als im Web-Client gesendet.

Diskussion der Messungen

Die mit dem Prototypen durchgeführten Messungen zeigen einige Unterschiede je nach Datenart, Lastszenario und Clienttyp (Browser vs. Konsolenclient). Während einige der im theoretischen Teil beschriebenen Annahmen zu den Unterschieden zwischen REST, GraphQL, gRPC und gRPC-Web bestätigt werden konnten, ergaben sich durch die Messungen folgende Erkenntnise:

- **Textdaten und Blogdaten im Browser:** Bei kleinen Textlasten und generell Datenstrukturen (Blog-Daten) sind GraphQL und gRPC-Web die schnellsten API-Architekturen. REST lag in fast allen Browser-Messungen leicht dahinter. Bei größeren Textlasten ist gRPC-Web am langsamsten. Es zeigt, dass gRPC-Web ab einer bestimmten Datenmenge nicht mehr performant ist. Der Versuch mit der hohen Parallelität bestätigte dieses Verhalten.
- **Mediendaten im Browser:** Für Mediendaten (Binärdaten im MB Bereich) zeigt sich REST in allen Kategorien als klar überlegen dar. GraphQL und gRPC-Web weisen ab einer gewissen Datenmenge sehr hohe Latenzen auf und können nicht mehr von allen Browserengines verarbeitet werden.
- **Browservergleich:** Der Browser beeinflusst die Ergebnisse spürbar. Es zeigen sich sowohl deutlich messbare Unterschiede in der Latenz als auch Mediendaten können von bestimmten Browsern besser bzw. schlechter verarbeitet werden. Die Messungen zeigen eine deutliche Abhängigekeit der Performance vom Webbrowser auf, Grund sind die unterschiedlichen Browserengines.
- **Konsolenclient (Microservice-Szenario):** Ohne Browser-Overhead und mit dem 'tatsächlichen' gRPC Protokoll, zeigt sich gRPC bis zu einer Datenmenge von 30 MB am performantesten, während REST auch noch bei sehr großen Dateien (Video) weiterhin gut funktionierte. gRPC-Web lag hier konstant hinter REST und gRPC. Die Messungen zeigten auf, dass es deutliche Unterschiede zwischen der Performance von gRPC und gRPC-Web Protokollen gibt.

Damit lässt sich festhalten, dass die Wahl der Architektur stark vom Einsatzzweck abhängt: während für Abfragen in Microservice-Architekturen gRPC bis zu einer gewissen Datenmenge am performantesten ist, stimmt dies nicht für gRPC-Web. gRPC-Web ist in Browser Umgebungen zwar bei kleineren Daten performant, bei etwas größeren Daten jedoch deutlich langsamer. Außerdem zeigen die Messungen, dass REST für Mediendaten eindeutig am besten geeignet ist, es klare Performacneunterschiede zwischen gRPC und gRPC-Web gibt und die Browserumgebung die Performance eindeutig beeinflusst.

Kapitel 6

Zusammenfassung und Fazit

Nachdem in den vorherigen Kapiteln theoretische Grundlagen, wissenschaftliche Arbeiten und ein selbst entwickelter Prototyp mit Messungen dazu eingesetzt wurden um Unterschiede der API-Architekturen zu ermitteln, werden in diesem Kapitel die Ergebnisse für die Beantwortung der Forschungsfragen zusammengeführt.

6.1 Beantwortung der Forschungsfragen

Forschungsfrage 1

Wie wirkt sich die Verwendung von gRPC bzw. gRPC-Web im Vergleich zu REST und GraphQL auf die Latenz, Effizienz und Ressourcennutzung in der Frontend-Backend-Kommunikation aus?

Latenz: Bezogen auf die Latenz werden der entwickelte Prototyp der die End-zu-End Latenz gemessen hat, und einige verwandte wissenschaftliche Arbeiten herangezogen. Die Messergebnisse des Prototyp zeigen, dass die Eignung der Architektur stark von de Datengröße abhängt. In Browserumgebungen, in denen nur gRPC-Web verwendet wird, erwiesen sich GraphQL und gRPC-Web bei kleinen Datengrößen (einige Bytes bis kB) am performantesten, wobei der Abstand zu REST mit 1-2 ms nicht sehr hoch ist. Werden jedoch größere Daten geschickt (200 kB), ist gRPC-Web mit einigen ms deutlich langsamer als GraphQL und REST. Bei der Übertragung von großen Binärdaten ist REST eindeutig die beste Wahl. Grund dafür ist, dass einerseits GraphQL wegen dem Base64-Overhead mehr Daten senden muss und andererseits Browser-Engines nativ auf die in REST verwendeten HTTP-Methoden ausgelegt sind. Auch die Chunk-Streaming Methode welche bei gRPC-Web für das Senden von

großen Daten implementiert wurde, konnte nicht effizient, und teilweise gar nicht, von den Browser Engines verarbeitet werden. Hierbei ist zu erwähnen, dass selbst falls es geeignetere Implementationen geben sollte, die Lösungswege wie solche große Datenmengen im Browser übertragen werden sollen kaum dokumentiert sind. REST ist diesbezogen auch bei den Punkten Erlernbarkeit und Dokumentation klar im Vorteil. Im Microservice-Szenario ohne Browser-Overhead erreichte gRPC die geringsten Latenzen, während REST bei sehr großen Dateien (z.B. Video mit 100 MB) weiterhin am effizientesten war. gRPC-Web blieb konstant hinter REST und gRPC zurück. Die Dominanz von gRPC gegenüber REST und GraphQL in Microservice-Architektur-Umgebungen konnte auch von zahlreichen verwandten Arbeiten in der theoretischen Analyse bestätigt werden, wobei in diesen Arbeiten auch gezeigt wurde, dass auch bei sehr kleinen Datenmengen im Byte Bereich, REST aufgrund von weniger Overhead schneller ist.

Effizienz: Hinsichtlich der Effizienz ergibt sich aus der theoretischen Analyse, dass gRPC und gRPC-Web aufgrund der Verwendung von Protobuf (und in Microservice Umgebungen auch aufgrund von HTTP/2-Mulitplexing) den geringsten Overhead aufweisen und vor allem bei vielen kleinen Nachrichten Vorteile bieten (Berg & Redi, 2023; Bolanowski u. a., 2022). REST und GraphQL sind bei einfachen Abfragen ähnlich effizient, wobei GraphQL durch gebündelte Abfragen das Problem des Over- bzw. Under-Fetchings vermeiden kann, was wiederum einen hohen Effizienzgewinn bringen könnte. Wie bereits erwähnt, ist GraphQL aufgrund der notwendigen Base64-Kodierung, welche mit rund einem Drittel zusätzlichem Overhead einhergeht deutlich weniger geeignet für große Binärdaten. Einige Browserengines, konnten diese GraphQL-Responses nicht verarbeiten. Bezogen auf Effizienz im Browserumfeld ist REST der klare Gewinner.

Ressourcennutzung: Bezüglich der Ressourcennutzung ergibt sich aus der theoretischen Analyse, dass gRPC und gRPC-Web durch das Protobuf-Parsing weniger CPU-Leistung als REST und GraphQL verbrauchen, da JSON-Parsing aufwändiger ist. GraphQL ist an sich am ressourvenintensivsten, kann jedoch durch die Flexibilität und das verhindern von Over- bzw. Under-Fetching wiederum Ressourcen einsparen (Niswar u. a., 2024).

Insgesamt zeigt sich, dass die Wahl der API-Architektur stark von der Datenmenge und der Umgebung abhängt. Während gRPC der klare Gewinner bezogen auf Latenz (abgesehen von sehr kleinen und sehr großen Daten), Ressourcennutzung und Effizienz ist, zeigt gRPC-Web in der Web-Umgebung bezogen auf diese Parameter kaum Vorteile. Ein Grund für die hohen Performanceeinbußen von gRPC zu gRPC-Web könnte,

abgesehen von dem Fehlen von gRPC-Features, damit erklärt werden, dass bei der Verwendung von gRPC-Web, Übersetzungsschritte notwendig sind, die gRPC in das gRPC-Web Protokoll umwandeln.

Forschungsfrage 2

Unter welchen Bedingungen ist der Einsatz von gRPC für die Frontend-Backend-Kommunikation sinnvoller als REST oder GraphQL?

Die Messergebnisse sowie die theoretische Analyse zeigen, dass sich der Einsatz von gRPC performancetechnisch vorallem in Service-zu-Service Architekturen lohnt. Die effiziente binäre Serialisierung mittels Protobuf und die Nutzung von HTTP/2 führen im Vergleich zu REST zu einer geringeren Latenz, höheren Effizienz und niedrigerem Ressourcenverbrauch. Dies kann insbesondere dann von großem Vorteil sein, wenn die Performance eine zentrale Rolle spielt (Berg & Redi, 2023; Bolanowski u. a., 2022; Niswar u. a., 2024).

Handelt es sich bei dem Frontendclient jedoch um einen Web-Client, kann gRPC nicht mehr verwendet werden, und es muss das eigens dafür entwickelte gRPC-Web Protokoll verwendet werden, bei welchen zentrale Vorteile von gRPC verloren gehen und zusätzliche Übersetzungsschritte erforderlich sind. Wie die Messergebnisse zeigen, lohnt sich der Einsatz von gRPC-Web aus Performancesicht nicht. Es lässt sich keine Verbesserung der Latenz feststellen, und im Vergleich zum etablierten REST, kommt es zu Problemen bei der Übertragung von größeren Mediendaten. Abgesehen davon ist die Erlernbarkeit im Vergleich zu etablierten Technologien wie REST deutlich komplexer und es stehen weniger Dokumentation, Tools und Community-Ressourcen zur Verfügung. GraphQL eignet sich im Frontend-Client besonders gut, um Over- und Underfetching zu vermeiden und es bestehen im Vergleich zu gRPC-Web deutlich mehr Ressourcen zur Verfügung, ist einfacher zu erlernen und performancetechnisch nicht schlechter. Aus diesen genannten Gründen wird geschlussfolgert, dass der Einsatz von gRPC-Web nur dann sinnvoll, wenn die bestehende Systemarchitektur bereits stark auf gRPC im Backend aufbaut und die Webintegration lediglich eine Erweiterung darstellt. Für einfache Abfragen sind REST und GraphQL im Browser jedoch die bessere Wahl.

Zusammenfassend ist gRPC-Web im Browserumfeld aus performancetechnischen Gründen nicht sinnvoller als REST oder GraphQL. Hier überwiegen die Nachteile, sodass REST und GraphQL die praktikableren Lösungen darstellen. Im Backend-Kontext hingegen ist gRPC sinnvoller, da es dort eine besonders effiziente, performante und ressourcenschonende Kommunikation ermöglicht.

6.2 Ausblick

Wie in der Arbeit feststellt, stellt gRPC-Web momentan keine eindeutige Alternative zu den etablierten Standards REST oder GraphQL für die Frontend zu Backendkommunikation in Webbrowsern dar. Mit der Weiterentwicklung moderner Webbrowser und einer somit umfassenden Unterstützung von HTTP/2 könnte sich die Leistungsfähigkeit von gRPC-Web jedoch verbessern. Eine steigende Popularität, könnte demnach dann ebenfalls zu mehr Dokumentation, Tools und Community-Ressourcen führen. Dadurch ist es möglich, dass gRPC in Zukunft performancetechnisch dennoch konkurrenzfähig zu REST und GraphQL im Browserkontext wird und damit auch für die Frontend-Backend-Kommunikation eine relevantere Rolle einnimmt.

Acronyms

Protobuf Protocol Buffers

JSON JavaScript Object Notation

REST Representational State Transfer

GraphQL Graph Query Language

gRPC gRPC Remote Procedure Call

HTTP Hypertext Transfer Protocol

API Application Programming Interface

Blob Binary Large Object

HTTPS Hypertext Transfer Protocol Secure

RPC Remote Procedure Call

Literaturverzeichnis

Amazon Web Services (2025). *Why Use GraphQL over REST?* Accessed: 2025-08-15. [online] https://docs.aws.amazon.com/appsync/latest/devguide/why-use-graphql.html (siehe S. 18, 23–25).

Amazon Web Services, Inc. (2024). *The difference between frontend and backend explained*. Accessed: 2025-06-09. [online] https://aws.amazon.com/de/compare/the-difference-between-frontend-and-backend/ (siehe S. 4).

– (2025). *What is Latency?* Accessed: 2025-06-09. [online] https://aws.amazon.com/de/what-is/latency/ (siehe S. 15).

AWS (2025). *RPC vs REST – Difference Between API Architectures*. Accessed: 2025-06-09. [online] https://aws.amazon.com/compare/the-difference-between-rpc-and-rest/ (siehe S. 12).

Berg, J. & D. M. Redi (Juni 2023). *Benchmarking the request throughput of conventional API calls and gRPC: A comparative study of REST and gRPC*. Bachelor's thesis TRITA-EECS-EX 2023:437. Stockholm, Sweden: KTH Royal Institute of Technology, School of Electrical Engineering und Computer Science (EECS). [online] https://kth.diva-portal.org/smash/record.jsf?pid=diva2:1792957 (siehe S. 21, 25, 26, 50, 51).

Bolanowski, M., K. Żak, A. Paszkiewicz, M. Ganzha, M. Paprzycki, P. Sowiński, I. Lacalle & C. E. Palau (Sep. 2022). „Efficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems". In: *Proceedings of the International Conference on Intelligent Software Methodologies, Tools and Techniques (SOMET 2022)*. Available via ResearchGate: https://www.researchgate.net/publication/362410022_Eficiency_of_REST_and_gRPC_realizing_communication_tasks_in_microservice-based_ecosystems. [online] https://www.researchgate.net/publica

tion/362410022_Eficiency_of_REST_and_gRPC_realizing_communication_tasks_in_microservice-based_ecosystems (siehe S. 20, 50, 51).

Brandhorst, J. (Jan. 2019). *The state of gRPC in the browser*. Blog post on gRPC.io. Accessed: 2025-06-09. [online] https://grpc.io/blog/state-of-grpc-web/ (siehe S. 1, 13).

Brito, G. & M. T. Valente (2020). „REST vs GraphQL: A Controlled Experiment“. In: *2020 IEEE International Conference on Software Architecture (ICSA)*. Accessed: 2025-08-15, S. 81–91. DOI: 10.1109/ICSA47634.2020.00016. [online] https://arxiv.org/abs/2003.04761 (siehe S. 26).

Charboneau, T. (2022). *6 Examples of GraphQL in Production at Large Companies*. Accessed: 2025-06-10. [online] https://nordicapis.com/6-examples-of-graphql-in-production-at-large-companies/ (siehe S. 18).

Chrome Developers (2025). *What is Blink? | Web Platform*. Accessed: 2025-08-31. [online] https://developer.chrome.com/docs/web-platform/blink (siehe S. 15).

Ecma International (Dez. 2017). *The JSON Data Interchange Syntax*. Standard, 2nd edition. JSON-Daten-Interchange-Syntax, veröffentlicht Dezember 2017. [online] https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf (siehe S. 5).

Fielding, R. T. (2000). „Architectural Styles and the Design of Network-based Software Architectures“. Accessed: 2025-06-12. Diss. University of California, Irvine. [online] https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (siehe S. 9, 23).

Google (2024). *Protocol Buffers Documentation*. Accessed: 2025-06-09. [online] https://protobuf.dev/overview/ (siehe S. 6, 12).

GraphQL Foundation (2025). *GraphQL Documentation*. Accessed: 2025-06-09. [online] https://graphql.org/learn/ (siehe S. 5, 11, 18, 24).

Group, I. H. W. (2022). *Hypertext Transfer Protocol (HTTP): Semantics*. RFC 9110. Standards Track, accessed 2025-08-31. [online] https://datatracker.ietf.org/doc/html/rfc9110 (siehe S. 10).

gRPC Authors (2025a). *About gRPC*. Accessed: 2025-08-20. [online] https://grpc.io/about/ (siehe S. 1, 12, 18, 25, 26).

– (2025b). *gRPC-Web Basics*. Accessed: 2025-06-09. [online] https://grpc.io/docs/platforms/web/basics/ (siehe S. 14, 25).

Kamiński, Ł., M. Kozłowski, D. Sporysz, K. Wolska, P. Zaniewski & R. Roszczyk (Dez. 2022). „Comparative review of selected Internet communication protocols“.

In: *arXiv preprint*. Preprint; DOI: 10.48550/arXiv.2212.07475. [online] https://www.researchgate.net/publication/366321051_Comparative_review_of_selected_Internet_communication_protocols (siehe S. 20).

Kubernetes Authors (2025). *Case Study: Spotify*. Accessed: 2025-08-15. [online] https://kubernetes.io/case-studies/spotify/ (siehe S. 19).

Ludwig-Mayerhofer, W. (2025). *Statistik I – Maßzahlen der zentralen Tendenz*. Accessed: 2025-08-15. [online] https://www.uni-siegen.de/phil/sozialwissenschaften/soziologie/mitarbeiter/ludwig-mayerhofer/statistik/statistik_downloads/statistik_i_3.pdf (siehe S. 15).

Microsoft Azure Architecture Center (Aug. 2025). *Best practices for RESTful web API design*. Documentation on Microsoft Learn. Accessed: 2025-06-15. [online] https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design (siehe S. 5).

Microsoft Docs (2025). *Use gRPC-Web with .NET*. Accessed: 2025-08-10. [online] https://learn.microsoft.com/en-us/aspnet/core/grpc/grpcweb?view=aspnetcore-9.0 (siehe S. 8, 14).

Mugur Marculescu (2015). *Introducing gRPC, a new open source HTTP/2 RPC Framework*. Posted: February 26, 2015; Accessed: 2025-06-10. [online] https://developers.googleblog.com/en/introducing-grpc-a-new-open-source-http2-rpc-framework/ (siehe S. 12).

Mühlberger, M. (2025). *Bachelorarbeit Prototyp (GitHub Repository)*. https://github.com/michaelmuehlberger/bachelorarbeit-prototyp/. Zugriff am 03.09.2025 (siehe S. 32).

Netflix Tech Blog (2023). *Practical API Design at Netflix, Part 1: Using Protobuf & FieldMask*. Accessed: 2025-08-31. [online] https://netflixtechblog.com/practical-api-design-at-netflix-part-1-using-protobuf-fieldmask-35cfdc606518 (siehe S. 19).

Niswar, M., R. A. Safruddin, A. Bustamin & I. Aswad (Juni 2024). „Performance evaluation of microservices communication with REST, GraphQL, and gRPC“. In: *International Journal of Electronics and Telecommunications* 70.2, S. 429–436. DOI: 10.24425/ijet.2024.149562. [online] https://www.researchgate.net/publication/381763921_Performance_evaluation_of_microservices_communication_with_REST_GraphQL_and_gRPC (siehe S. 21, 25, 26, 50, 51).

Postman (2022). *2022 State of the API Report*. Accessed: 2025-08-15. [online] https://www.postman.com/state-of-api/2022/ (siehe S. 17, 22, 24).

– (2023). *2023 State of the API Report*. Accessed: 2025-06-09. [online] https://www.postman.com/state-of-api/2023/ (siehe S. 17, 22, 24, 26).

Postman Team (2023). *What Is a REST API? Examples, Uses & Challenges*. Accessed: 2025-06-09. [online] https://blog.postman.com/rest-api-examples/ (siehe S. 18).

Red Hat, Inc. (2020). *An architect's guide to APIs: SOAP, REST, GraphQL, and gRPC*. Accessed: 2025-08-20. [online] https://www.redhat.com/en/blog/apis-soap-rest-graphql-grpc (siehe S. 2, 8, 23–26).

Roblox Newsroom (2025). *Roblox's Path to 2 Trillion Analytics Events a Day*. Accessed: 2025-08-15. [online] https://corp.roblox.com/newsroom/2025/06/roblox-path-to-2-trillion-analytics-events-a-day (siehe S. 19).

Stack Overflow (2023). *Stack Overflow Developer Survey 2023 – Technology page*. Accessed: 2025-08-20. [online] https://survey.stackoverflow.co/2023/ (siehe S. 17).

Thomson, M. & M. Bishop (2022). *Hypertext Transfer Protocol Version 2 (HTTP/2)*. Internet Engineering Task Force (IETF) RFC 9113. Accessed: 2025-06-15. [online] https://datatracker.ietf.org/doc/html/rfc9113 (siehe S. 7).

World Wide Web Consortium (W3C) (2024). *File API*. W3C Working Draft, accessed 2025-08-20. [online] https://www.w3.org/TR/FileAPI/ (siehe S. 7).

Anhang A

Anhang

A.1 Rohdaten der Messungen

Im Folgenden sind die Rohdaten aller durchgeführten Messreihen des Prototyps aufgeführt.

Tabelle A.1: Rohdaten: Web-Client Einzelrequests - Chrome (Antwortzeit in ms; 30 Messungen je Kombination)

Technologie	Service	Datentyp	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
REST	Text	small text	9.2	9.7	8.2	8.7	7.3	7.9	11.9	8.4	9.5	7.4	9.2	10.1	9.4	7.7	9.2	8.7	9.7	9.2	7.5	8.7	7.5	7.9	7.3	8.8	7.4	9.3	7.9	7.7	8.9	7.6
REST	Text	medium text	8.6	7.6	9.6	8.7	7.8	8.8	9.0	7.7	9.8	9.5	8.8	9.0	8.6	8.5	9.8	12.0	9.5	8.0	9.4	7.6	9.2	8.3	8.7	7.9	8.8	7.8	10.0	8.7	8.5	13.9
REST	Text	large text	13.4	15.0	12.2	13.3	12.2	13.5	20.2	14.1	13.9	12.5	12.8	13.0	15.1	11.7	14.1	12.8	14.4	12.2	14.5	14.9	12.3	11.7	11.8	13.1	9.3	12.8	12.3	14.2	11.0	12.3
REST	Media	Foto	22.4	44.4	26.0	34.2	25.1	36.3	32.3	26.7	45.9	30.2	36.8	38.0	36.1	37.3	29.6	40.5	35.2	32.5	41.3	36.6	37.9	40.6	41.0	34.7	35.3	43.5	44.5	34.7	45.8	38.4
REST	Media	Audio	175.3	150.5	130.4	125.0	144.4	148.3	141.4	127.3	129.9	136.8	158.2	156.5	136.8	152.8	139.2	131.7	155.1	155.6	161.5	161.8	174.5	140.5	164.5	152.6	156.6	175.5	155.4	158.7	127.2	143.5
REST	Media	Video	353.2	392.7	399.9	398.6	337.2	341.2	332.7	401.2	373.2	312.8	392.9	343.7	339.8	376.2	351.7	340.6	416.8	405.1	366.4	354.0	372.5	352.8	336.2	386.0	346.6	346.2	342.4	400.3	386.5	351.3
REST	Blog	-	8.2	7.9	7.4	13.9	8.2	8.6	7.8	7.5	8.2	8.5	9.5	7.7	10.2	7.7	10.6	10.3	8.6	8.6	8.7	7.8	10.1	7.8	8.0	9.5	7.7	7.3	8.1	8.7	7.8	7.5
GraphQL	Text	small text	7.5	7.1	5.5	10.2	6.6	6.5	7.5	8.3	6.8	10.9	7.4	7.0	7.4	7.5	7.4	8.6	6.8	6.8	6.6	7.7	7.6	10.6	8.4	7.2	7.8	7.9	6.5	7.4	7.0	8.3
GraphQL	Text	medium text	7.8	7.9	7.4	8.3	9.2	7.7	7.5	7.3	8.4	8.0	7.6	8.1	7.9	10.7	7.4	7.4	7.5	7.8	6.3	7.4	7.6	6.7	7.6	6.6	7.7	8.3	6.7	7.3	7.1	7.4
GraphQL	Text	large text	12.5	10.5	10.8	9.1	11.4	9.5	7.7	10.0	10.8	8.5	11.5	11.0	8.9	10.3	9.4	10.1	9.6	10.5	11.4	7.6	16.4	7.9	10.5	9.2	8.8	13.2	9.1	9.2	9.3	9.1
GraphQL	Media	Foto	378.7	671.5	418.5	391.1	459.3	403.5	534.4	536.3	409.3	589.7	669.9	703.8	657.1	424.3	430.7	353.3	521.7	363.3	665.4	419.6	396.6	442.4	412.4	351.4	602.8	657.1	577.1	665.0	688.6	557.5
GraphQL	Media	Audio	5441.9	6626.5	5337.3	7689.2	7138.3	5708.4	5119.7	6504.3	4999.1	5662.9	5683.2	6117.4	5028.2	5300.6	5138.4	5000.3	5829.4	7507.8	5416.1	5653.7	6113.1	4930.8	5319.7	7180.0	5168.1	5221.8	7857.7	5452.2	6798.7	5961.7
GraphQL	Media	Video															-															
GraphQL	Blog	-	5.9	7.5	10.7	8.1	7.0	6.7	6.3	6.9	6.3	6.9	8.5	7.1	7.3	10.3	6.3	6.9	8.4	7.7	7.1	7.5	8.0	6.6	7.5	10.7	6.4	7.9	6.7	9.6	9.1	6.4
gRPC	Text	small text	7.2	7.6	7.7	6.8	7.3	7.3	6.9	7.5	10.2	7.5	8.9	7.5	7.9	6.9	7.2	8.5	5.3	7.8	7.6	7.1	11.2	7.6	7.0	6.5	7.4	7.8	6.6	7.6	6.5	6.6
gRPC	Text	medium text	8.2	8.7	8.5	10.5	8.4	8.5	8.0	7.5	7.6	8.8	11.4	8.1	8.3	9.5	7.8	8.3	7.5	8.6	8.7	7.7	9.1	8.2	7.1	8.0	7.3	8.5	8.3	7.0	8.6	7.9
gRPC	Text	large text	16.9	19.7	19.6	21.9	17.6	21.6	19.0	19.2	18.6	17.7	17.2	19.0	15.4	18.8	17.0	20.2	18.2	18.8	18.4	17.7	19.3	22.7	20.1	19.9	16.2	18.8	19.1	22.0	16.8	16.3
gRPC	Media	Foto	128.4	121.1	193.0	125.8	158.4	243.7	160.0	248.2	170.5	182.3	239.6	151.9	278.7	182.4	154.0	158.4	364.9	155.3	277.4	165.2	142.5	156.8	215.6	169.4	165.9	116.3	120.3	148.3	211.3	153.7
gRPC	Media	Audio	2283.3	2643.5	1726.5	2356.3	2366.4	2107.2	1967.7	2393.3	1924.8	2390.3	1974.8	1606.8	2506.9	2374.6	2145.8	2306.7	2575.3	2109.4	2168.8	2501.5	3549.0	3192.8	2165.9	2942.9	2431.4	2425.1	2353.6	2308.2	3079.1	2295.9
gRPC	Media	Video															-															
gRPC	Blog	-	7.7	8.3	7.5	7.6	7.8	6.4	7.0	7.7	7.0	8.7	6.7	8.2	6.2	8.1	7.5	11.9	7.9	7.4	8.3	6.8	7.9	7.2	8.1	7.3	8.3	7.8	6.5	7.9	12.3	8.2

Tabelle A.2: Rohdaten: Web-Client *20 parallele Requests* - Chrome (Antwortzeit in ms; bis zu 30 Messungen je Kombination)

Technologie	Service	Datentyp	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
REST	Text	small text	45.8	52.4	42.7	30.2	37.6	37.7	35.0	30.7	54.5	38.1	45.3	51.4	33.6	47.8	30.2	76.6	30.4	57.0	59.5	32.4	23.6	69.6	63.1	33.8	42.0	58.2	68.4	51.4	45.6	40.1
REST	Text	medium text	61.7	48.9	55.6	82.8	50.2	84.3	64.0	35.8	63.2	70.7	57.6	52.6	58.3	47.0	83.5	62.2	42.6	49.4	62.4	87.0	68.0	59.7	72.5	41.9	79.1	52.2	67.1	36.9	84.4	74.0
REST	Text	large text	113.6	139.5	100.8	76.7	109.6	94.5	98.7	89.0	84.3	102.4	82.0	79.7	94.0	100.8	98.2	110.3	107.5	69.6	85.3	80.7	87.1	93.2	64.2	96.3	106.0	61.6	83.7	96.1	73.5	84.1
REST	Media	Foto	360.9	453.2	370.0	423.0	415.8	399.2	400.4	362.4	368.5	304.4										-										
REST	Media	Audio	2770.0	2400.8	2282.5	2350.9	2391.0	2367.5	2361.6	2450.4	2395.7	2430.4										-										
REST	Media	Video												-																		
REST	Blog	-	72.6	72.2	62.8	39.2	74.6	42.4	42.4	83.6	44.9	30.5	43.9	42.2	60.1	50.6	43.3	69.4	65.5	34.1	60.9	69.6	36.2	39.6	61.0	60.3	32.7	32.7	45.9	40.3	71.9	65.1
GraphQL	Text	small text	26.2	32.8	26.8	26.2	29.1	33.8	43.7	29.9	28.9	33.4	34.8	35.5	31.4	29.5	27.3	30.3	23.7	54.7	26.4	26.3	25.6	27.9	35.2	27.6	28.7	26.0	26.8	46.3	27.4	41.1
GraphQL	Text	medium text	32.3	27.7	35.3	35.6	54.5	39.7	58.5	38.0	28.9	26.1	45.7	39.1	29.1	30.5	33.1	33.6	31.2	27.8	29.0	27.7	49.5	31.8	28.2	30.0	40.9	34.0	33.6	31.2	33.8	44.7
GraphQL	Text	large text	61.2	75.4	59.6	83.2	73.2	64.3	59.0	81.5	72.9	65.9	58.9	82.4	77.5	83.3	69.7	83.3	67.7	53.2	60.4	71.5	85.0	66.9	63.0	62.5	84.3	99.0	64.5	87.6	66.7	50.9
GraphQL	Media	Foto	17657.2	19009.6	16089.2	15402.1	19715.3	14381.0	13484.4	16510.5	14588.1	14758.3										-										
GraphQL	Media	Audio												-																		
GraphQL	Media	Video												-																		
GraphQL	Blog	-	42.3	68.2	36.3	58.7	32.7	29.2	33.4	38.1	48.2	41.1	35.8	26.5	17.8	35.3	25.4	28.1	29.8	30.2	39.6	27.3	24.4	23.2	29.4	36.6	28.3	43.9	28.7	26.4	33.3	40.0
gRPC	Text	small text	26.4	40.1	28.5	29.3	51.3	39.7	26.4	35.2	30.9	24.4	40.5	33.3	33.4	26.8	21.3	46.9	29.5	17.3	25.3	30.1	38.7	44.3	41.9	30.5	29.5	43.4	39.4	34.1	33.2	30.7
gRPC	Text	medium text	39.5	39.2	46.1	39.2	43.6	65.2	46.6	36.8	42.1	56.4	49.5	36.3	32.3	45.1	35.7	34.7	28.8	35.5	34.6	37.8	45.8	44.9	63.3	65.1	49.7	57.2	48.9	39.6	35.1	30.6
gRPC	Text	large text	135.9	132.6	130.3	145.3	143.8	146.4	121.6	152.5	171.1	128.5	143.2	141.4	156.8	139.6	146.9	148.7	157.8	145.6	147.5	151.6	145.5	145.2	163.5	127.7	132.2	169.1	147.1	141.4	141.1	158.5
gRPC	Media	Foto	5423.3	6538.4	5350.4	5636.5	5728.3	6121.5	5925.4	5520.5	5293.3	5824.7										-										
gRPC	Media	Audio	100343.2	101637.3	103693.2	108612.8	161328.4	233593.4	110382.3	107263.2	103128.3	130283.2										-										
gRPC	Media	Video												-																		
gRPC	Blog	-	29.2	30.8	30.4	52.4	34.4	26.2	41.8	31.0	29.2	37.8	35.4	27.8	39.2	31.5	38.6	28.4	39.0	42.4	28.8	39.5	28.2	28.0	36.9	28.2	48.8	30.7	45.3	37.3	29.8	28.6

Tabelle A.3: Rohdaten: Web-Client Einzel-Request Firefox (Antwortzeit in ms; 10 Messungen je Kombination)

Technologie	**Service**	**Datentyp**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
REST	Text	small text	8	9	5	3	7	8	12	10	16	7
REST	Text	medium text	8	8	8	9	15	15	8	9	7	6
REST	Text	large text	13	12	10	12	11	10	10	15	10	11
REST	Media	Foto	29	63	43	77	50	49	54	55	56	68
REST	Media	Audio	269	324	222	238	247	243	257	214	224	249
REST	Media	Video	701	649	688	677	702	631	680	648	717	671
REST	Blog	-	7	5	4	7	14	5	6	6	7	6
GraphQL	Text	small text	7	14	7	7	15	10	7	8	11	8
GraphQL	Text	medium text	13	7	8	9	16	12	8	12	10	16
GraphQL	Text	large text	29	15	10	12	11	23	14	19	10	15
GraphQL	Media	Foto	2099	1691	1736	1667	1684	1689	1799	1688	1709	1672
GraphQL	Media	Audio	13638	14587	13308	12725	13592	12436	13036	12866	13221	13124
GraphQL	Media	Video	47634	49843	46806	52938	46273	40856	44876	48937	472374	50373
GraphQL	Blog	-	11	16	7	11	9	7	5	9	14	7
gRPC	Text	small text	11	11	7	8	15	6	13	7	6	10
gRPC	Text	medium text	11	7	9	12	14	9	9	6	7	10
gRPC	Text	large text	11	24	27	28	26	29	26	20	31	20
gRPC	Media	Foto	11	260	265	292	236	277	293	269	321	257
gRPC	Media	Audio	11	2723	2837	2800	2797	2677	2529	3000	2616	2566
gRPC	Media	Video	11	12029	9728	9500	10178	9506	10608	9923	10191	8174
gRPC	Blog	-	11	10	7	6	8	9	7	7	8	7

Tabelle A.4: Rohdaten: Web-Client Einzel-Request Edge (Antwortzeit in ms; 10 Messungen je Kombination)

Technologie	**Service**	**Datentyp**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
REST	Text	small text	5.4	5	5.6	4.8	5.4	3.7	4.5	3.7	4.9	5.4
REST	Text	medium text	7.7	5.6	6.6	5.3	6.2	5.6	6	5.9	6.8	7.4
REST	Text	large text	7.2	10.5	9.5	7.6	9.2	9.8	10.3	10.3	11.1	9.5
REST	Media	Foto	36.7	23.4	39	38.1	38.6	53.6	33.8	24.7	35.9	39.1
REST	Media	Audio	159.2	148.8	168.3	136.6	138.9	150.9	128.2	140.4	146.3	131.2
REST	Media	Video	524	342	458.5	466.3	413	384.2	398.8	395.6	373.2	472.3
REST	Blog	-	7.5	8.5	7.4	7.7	8.5	8.4	7.8	8.7	7.2	7.1
GraphQL	Text	small text	5.5	7.7	5.2	6.2	5.2	5.7	5.2	5.5	5.1	4.8
GraphQL	Text	medium text	6.3	4.4	5.1	4.9	6	6	5.5	7.2	5.5	5.9
GraphQL	Text	large text	9.3	8.3	6.5	7	7	12.2	7.4	10.2	6.9	7.3
GraphQL	Media	Foto	376.6	403.9	399.4	389.8	370.5	455	526.2	384.1	355	367
GraphQL	Media	Audio	4803.3	9124	6786	6241.3	6525	6942.1	4940.9	6190.2	5521.3	7346.6
GraphQL	Media	Video					-					
GraphQL	Blog	-	8.5	4.3	5.6	4.1	5.8	4.5	7.9	4.9	5.4	4.9
gRPC	Text	small text	5.4	4	5.7	7.7	5.3	5.1	4.9	5.7	4.4	6.8
gRPC	Text	medium text	7	5.6	6.3	5.7	5.1	4.5	5.9	5.1	6.1	5.6
gRPC	Text	large text	17.7	19.1	25.8	23.9	21.5	23.6	17.3	16.9	16.8	20.5
gRPC	Media	Foto	216.5	172.5	236.7	179.7	215.5	177.9	139.3	174.4	133.1	218.3
gRPC	Media	Audio	4060.4	2417.1	2734.9	2471.9	2219.3	2523.1	2849.8	2518.3	3002.1	2652.4
gRPC	Media	Video	7957.2	8901.1	7430.3	8105.8	7471.8	8650.3	7792.9	7561.1	7745.5	8230.2
gRPC	Blog	-	5	4.8	4.9	5	5	5.5	5	6.4	5.3	5.3

Tabelle A.5: Rohdaten: Konsolen-Client (Microservice) - Einzel-Requests (Antwortzeit in ms; 10 Messungen je Kombination)

Technologie	**Service**	**Datentyp**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
REST	Text	small text	2	3	2	3	2	2	2	2	3	2
REST	Text	medium text	3	2	2	2	2	2	2	2	2	2
REST	Text	large text	7	7	7	7	7	10	3	6	6	4
REST	Media	Foto	80	52	39	48	59	34	44	30	37	36
REST	Media	Audio	252	260	298	252	247	282	240	230	213	230
REST	Media	Video	680	688	500	556	481	573	458	530	472	464
REST	Blog	-	1	1	1	1	1	1	1	1	1	1
GraphQL	Text	small text	3	2	2	2	2	2	3	2	2	3
GraphQL	Text	medium text	2	2	2	2	3	2	2	2	2	3
GraphQL	Text	large text	5	6	5	7	6	5	5	5	4	7
GraphQL	Media	Foto	4658	5972	6171	5790	5694	6102	5611	5591	5966	5607
GraphQL	Media	Audio					-					
GraphQL	Media	Video					-					
GraphQL	Blog	-	3	2	2	2	3	3	2	2	3	2
gRPC-Web	Text	small text	2	3	2	3	2	2	2	2	2	2
gRPC-Web	Text	medium text	3	3	3	3	2	3	3	2	3	3
gRPC-Web	Text	large text	7	7	6	7	7	7	7	6	7	7
gRPC-Web	Media	Foto	55	50	40	37	53	55	53	53	69	48
gRPC-Web	Media	Audio	267	238	254	263	245	244	275	236	251	241
gRPC-Web	Media	Video	625	649	696	729	745	750	717	718	702	720
gRPC-Web	Blog	-	3	2	3	3	3	3	3	3	3	3
gRPC	Text	small text	1	1	1	1	2	2	1	1	1	1
gRPC	Text	medium text	1	2	1	2	2	1	2	2	1	1
gRPC	Text	large text	3	4	4	4	4	4	4	5	5	4
gRPC	Media	Foto	38	40	52	38	44	43	40	39	51	42
gRPC	Media	Audio	217	248	292	241	257	243	250	238	251	284
gRPC	Media	Video	624	609	632	621	605	582	625	583	589	630
gRPC	Blog	-	1	1	1	1	2	1	1	1	1	1